Methodology and Tools for Energy-aware Task Mapping on Heterogeneous Multiprocessor Architectures

Baptiste Roux

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du signal et Télécommunications

Ecole doctorale MathSTIC

présentée par

Baptiste Roux

préparée à l'unité de recherche IRISA (UMR 6074)
Institut de Recherche en Informatique et Systèmes Aléatoires
École Nationale Supérieure des Sciences Appliquées et de Technologie

---

Methodology and Tools for Energy-aware Task Mapping on Heterogeneous Multiprocessor Architectures

Thèse soutenue à Rennes le 23 novembre 2017

devant le jury composé de :

Virginie FRESSE
MCF - Université Jean Monnet / rapporteur

Frédéric ROUSSEAU
PR - Université Grenoble Alpes / rapporteur

Jean-Philippe DELAHAYE
Ingénieur - DGA MI / examinateur

Bertrand GRANADO
PR - Université Pierre et Marie Curie / examinateur

Loic LAGADEC
PR - ENSTA Bretagne / examinateur

Matthieu GAUTIER
MCF - Université Rennes 1 / co-directeur de thèse

Olivier SENTIEYS
PR - Université Rennes 1 / directeur de thèse

Résumé

Au cours de la dernière décennie, la conception des systèmes embarqués a évolué dans l'optique d'augmenter la puissance de calcul tout en conservant une faible consommation d'énergie. À titre d'exemple, les véhicules autonomes tels que les drones sont un domaine d'application représentatif qui combine de la vision, des communications sans fil avec d'autres noyaux de calculs intensifs, le tout avec un budget énergétique limité. Avec l'avènement des systèmes multicœurs sur puce (MpSoC), la simplification des processeurs a diminué la consommation d'énergie par opération, alors que leur multiplication a amélioré les performances. Cependant, l'apparition du phénomène de *dark silicon* a conduit à l'intégration d'accélérateurs matériels spécialisés au sein des systèmes multicœurs. C'est ainsi que sont nées les architectures massivement multicœurs hétérogènes (HMpSoC) combinant des processeurs généralistes (SW) et des accélérateurs matériels (HW). Pour ces architectures hétérogènes, les performances et la consommation d'énergie dépendent d'un large ensemble de paramètres tels que le partitionnement HW/SW, le type d'implémentation HW et le coût de communication entre les organes de calcul HW et SW conduisant ainsi à un immense espace de conception.

Dans cette thèse, nous étudions des méthodes permettant la réduction de la complexité de développement et de mise en œuvre d'applications efficaces en énergie sur HMpSoC. De nombreuses contributions sont proposées pour améliorer les outils d'exploration de l'espace de conception (DSE) avec des objectifs énergétiques. Tout d'abord, une définition formelle de la structure HMpSoC est introduite ainsi qu'une méthode de représentation générique axée sur la hiérarchie mémoire. Ensuite, un outil de modélisation rapide de l'énergie est proposé et validé sur plusieurs applications. Ce modèle énergétique sépare les sources d'énergie en trois catégories (calcul statique, dynamique et communications) et calcule leurs contributions sur la consommation globale de manière indépendante. Basée sur une étude précise des communications, cette approche calcule rapidement la consommation d'énergie pour une répartition donnée d'application sur un HMpSoC. Dans un deuxième temps, nous proposons une méthodologie permettant l'exploration énergétique d'accélérateurs sur HMpSoC. Cette méthode s'appuie sur le modèle de consommation précédent couplé à une formulation de programmation linéaire en nombre entier mixte (MILP). Cela permet de sélectionner efficacement les accélérateurs HW et le partitionnement HW/SW et ainsi d'obtenir une implémentation efficace en énergie pour une application tuilée. Les expériences réalisées ont montré la complexité du processus de validation d'outils/algorithmes de DSE sur une large gamme d'applications et d'architectures. Afin de résoudre ce problème, nous proposons un simulateur d'architectures HMpSoC intégrant un modèle de consommation permettant d'observer l'exécution d'applications. La structure de l'architecture cible est décrite à l'aide d'un fichier de configuration basé sur le modèle de représentation générique précédent. Ce fichier est chargé dynamiquement lors du démarrage du simulateur. De plus, ce simulateur est associé à un générateur d'applications permettant la création d'un large ensemble d'applications représentatives du domaine. Ce générateur se base sur un ensemble de schémas de calcul et de communication élémentaire qu'il combine pour obtenir une application complète. Les applications ainsi obtenues peuvent être enrichies par des informations de placement et automatiquement exécutées sur le simulateur. Cet ensemble d'outils a pour objectif de faciliter la validation de nouveaux algorithmes ciblant le placement efficace en énergie d'application sur une large gamme d'architectures HMpSoC.

Abstract

During the last decade, the design of embedded systems was pushed to increase computational power while maintaining low energy consumption. As an example, autonomous vehicles such as drones are a representative application domain which combines vision, wireless communications and other computation intensive kernels constrained with a limited energy budget. With the advent of Multiprocessor System-on-Chip (MpSoC) architectures, simplification of processor cores decreased power consumption per operation, while the multiplication of cores brought performance improvement. However, the *dark silicon* issue led to the benefit of augmenting programmable processors with specialized hardware accelerators and to the rise of Heterogeneous MpSoC (HMpSoC) combining both software (SW) and hardware (HW) computational resources. For these heterogeneous architectures, performance and energy consumption depend on a large set of parameters such as the HW/SW partitioning, the type of HW implementation or the communication cost between HW and SW cores therefore leading to a huge design space.

In this thesis, we study how to reduce the development and implementation complexity of energy-efficient applications on HMpSoC. Multiple contributions are proposed to enhance Design Space Exploration (DSE) tools with energy objectives. First, a formal definition of HMpSoC structure is introduced alongside with a generic representation focused on the memory hierarchy. Then, a fast power modelling tool is proposed and validated on several applications. This power model separates the power sources in three families (static, dynamic computation and dynamic communication) and computes their contributions on global consumption independently. With a fine grain communications study, this approach rapidly computes energy consumption for a given application mapping on a HMpSoC. In a second time, we propose a methodology for energy-driven accelerator exploration on HMpSoC. This method builds upon the previous power model coupled with an Mixed Integer Linear Programming (MILP) formulation and enables to efficiently select HW accelerators and HW/SW partitioning which achieve energy efficient-mapping of a tiled application. The experiments involved in these contributions show the complexity of DSE validation process on a wide range of applications and architectures. To address these issues, we introduce a HMpSoC simulator embedding a power model to monitor application execution. Properties of targeted architectures are described, at run-time with the previous generic representation model. Furthermore, this simulator is coupled with an application generator framework that could build an infinite set of representative applications following predefined computation models. The obtained applications could then be enriched with mapping directive and executed on the simulator. This combination enables to ease the research and validation of new DSE algorithms targeting energy-aware application mapping on a wide range of HMpSoC architectures.

Contents

Chapitre **ZÉRO**

RÉSUMÉ ÉTENDU

1 Contexte général

Les systèmes embarqués ont par définition des ressources énergétiques limitées. Un des verrous à l'augmentation des débits des objets communicants et embarqués est de pouvoir réaliser un nombre important de traitements sous cette contrainte d'énergie finie ou de puissance dissipée maximale. La mise en œuvre de la liaison de données entre les drones et une station de base terrestre est une application qui entre dans ce contexte. En effet, la gestion de l'énergie est un verrou limitant la durée des missions de ces drones. Ainsi, réduire la consommation des traitements numériques embarqués est un challenge important.

Cette thèse a été cofinancée par l'Institut National de Recherche en Informatique et en Automatique (INRIA) et la Direction Générale de l'Armement (DGA). Cette étude est née de besoins inhérents aux applications embarquées dans les drones de la DGA. Ces derniers utilisent un lien radio basé sur le standard DVB-S2 avec une architecture matérielle standard basée sur l'association d'un FPGA, d'un DSP et d'un processeur généraliste. L'utilisation de ces trois composants distincts complexifie le processus de reconfiguration et diminue les performances énergétiques globales. Cette thèse a pour but d'explorer les opportunités offertes par les nouvelles architectures hétérogènes associant processeurs généralistes et accélérateurs matériels avec le paradigme de communications reconfigurables basé sur la radio logicielle. L'objectif premier est de mettre en avant les forces et les faiblesses de ces architectures et de soulever les difficultés liées à la conception d'applications efficaces en énergie sur ces architectures hétérogènes. À long terme, nous voulons proposer une formalisation de ces architectures ainsi que des outils et des méthodes facilitant le développement d'applications économes en énergie sur ces architectures.

Dans ce chapitre, nous résumons le contexte de l'étude ainsi que les différentes contributions apportées au cours de la thèse. Dans les sous sections suivantes, nous présentons le concept de radio logicielle et les différentes avancées dans le monde de la microélectronique ayant entraîné l'apparition des architectures hétérogènes. La section 2 introduit la structure générique des architectures considérées ainsi qu'une méthode de représentation mettant l'accent sur les canaux de communications. Nous présentons également le format de représentation utilisé pour la description d'application. Enfin, nous introduisons un flot de développement orienté sur l'énergie ainsi que les travaux réalisés au vu de son adoption. Pour finir, la section 4 résume les différents éléments présentés dans ce document.

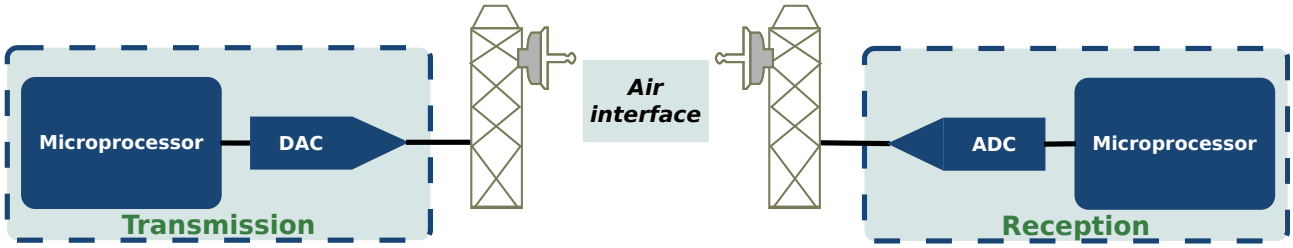


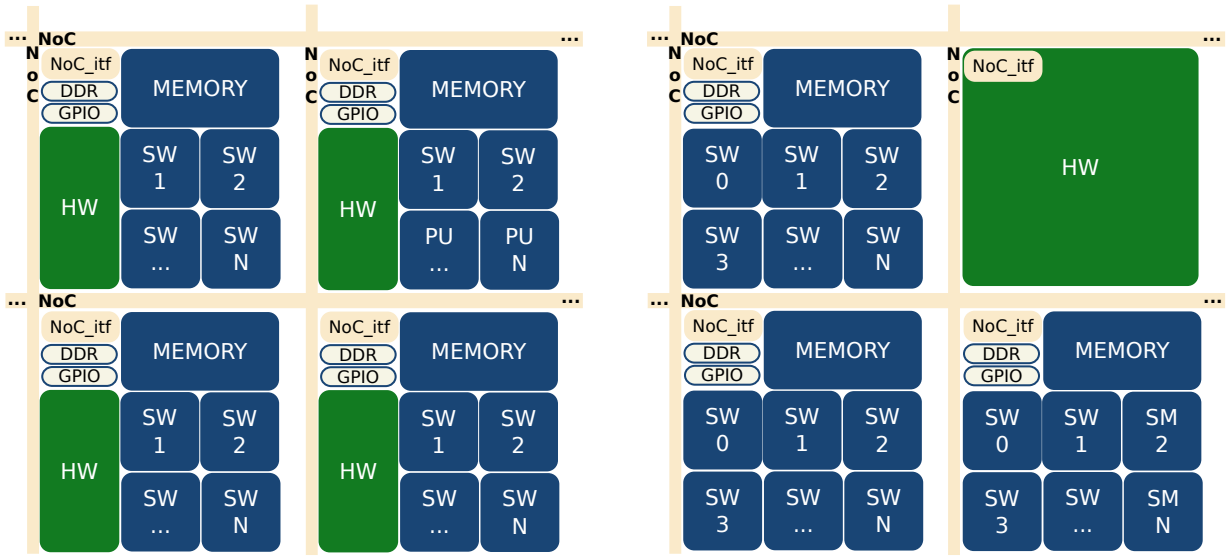
Figure 0-1 – **Structure globale d'une radio logicielle idéale.**

1.1 Radio logicielle

Le concept de radio logicielle trouve ses fondements dans le domaine militaire avec la nécessité d'assurer l'inter-opérabilité des équipements à l'aide de plateformes pouvant générer plusieurs formes d'ondes et normes de communication grâce à une simple reprogrammation. En effet, le cycle de vie des applications militaires est assez long et il est essentiel de les maintenir et d'assurer leur inter-opérabilité avec les applications à venir. Ce concept a été introduit pour la première fois dans les années 90 par Joseph Mitola [Mit93]. Il consiste à réduire la partie analogique à son strict minimum pour avoir plus de flexibilité de reconfiguration avec la partie numérique. Une architecture de radio logicielle idéale serait composée d'un convertisseur placé directement derrière l'antenne et d'un composant numérique programmable tel qu'un microprocesseur dédié aux traitements. Cela est illustré sur la Figure 0-1. Cette structure, permettant une flexibilité totale, est l'objectif ultime d'un système de radio logicielle. Malheureusement, numériser le signal juste après l'antenne requière une haute fréquence d'échantillonnage des convertisseurs et représente un frein. De plus, les implémentations de radio logicielle basées sur des microprocesseurs généralistes présentent certaines limites notamment liées aux performances énergétiques et aux débits atteignables. Néanmoins, la plupart des algorithmes utilisés est hautement parallèle et nécessite des performances de calculs importantes qui pourraient tirer profit des nouvelles architectures multicœurs hétérogènes et ainsi traiter les problèmes de consommation d'énergie au prix d'une programmation plus complexe.

1.2 Tendances architecturales

Par le passé, la loi de Moore [Moo65] prédisant le doublement du nombre de transistors sur puce tous les 18 mois, a été un moteur fondamental de l'évolution des microarchitectures. Associée à la loi de Dennard [Den+74] prédisant la diminution de la tension et du courant d'alimentation des transistors en proportion de leur taille, une série de circuits, d'architectures et de compilateurs a émergé, conduisant ainsi à une augmentation exponentielle des performances. Ces dernières ont été obtenues en passant progressivement d'architectures simple-cœur à des architectures multicœurs pour tirer pleinement profit du nombre de transistors disponibles. Malheureusement, un nouveau phénomène appelé "dark silicon" lié à la limitation de la densité de puissance dissipée sur une puce [Esm+11] est apparu. Ce dernier empêche l'utilisation simultanée de l'ensemble des transistors présents sur la puce et a poussé les concepteurs d'architectures à intégrer des accélérateurs matériels au sein des microarchitectures. C'est ainsi que sont nées les architectures multicœurs hétérogènes intégrant plusieurs cœurs processeurs et des accélérateurs matériels. Outre le gain en performance, ces architectures ont induit de nouvelles problématiques en termes de développement. En effet, avec la multiplication du nombre et du type d'organes de calcul, l'espace d'exploration est devenu immense. Ainsi, lorsque la consommation devient un facteur clé, il est essentiel de pouvoir explorer cet espace au plus tôt dans le flot de développement pour obtenir des solutions efficaces en énergie.



(a) **HMpSoC distribué.** (b) **HMpSoC partagé.**

Figure 0-2 – **Structure des architectures HMpSoC.**

2 Architectures et applications cibles

Dans cette section, nous présentons le formalisme utilisé dans la suite du document pour décrire les architectures et les applications utilisées.

2.1 Architectures cibles

Pour que notre étude puisse être appliquée à un large panel d'architectures, nous avons commencé par décrire de manière générique la structure des architectures ciblées. Pour cela, nous avons étudié diverses architectures utilisées dans le domaine de la radio logicielle ainsi que des architectures plus généralistes. Les architectures multicœurs sont généralement composées d'un ensemble de mémoires, de processeurs, d'éléments d'interconnexion et d'entrées/sorties. Lorsqu'elles intègrent des accélérateurs matériels, elles deviennent hétérogènes dans le sens où elles associent des processeurs généralistes (SW) et des accélérateurs matériels (HW). Une représentation générique de ces architectures, appelées HMpSoC, est proposée dans la Figure 0-2. Elle est construite à l'aide de clusters connectés par un réseau sur puce (NoC). Chaque cluster est composé de N cœurs de calcul logiciels associés à un accélérateur matériel de taille S. Au niveau du cluster, les communications sont effectuées grâce à une mémoire partagée. Suite à cette description, deux familles de HMpSoC peuvent être construites en fonction du placement des accélérateurs matériels dans l'architecture. Dans le cas où ils sont placés au sein de chaque cluster ($S \neq 0$), on obtient des *HMpSoC distribués* (Figure 0-2a). Ils permettent d'obtenir des communications rapides entre la partie matérielle et la partie logicielle mais en conséquence, la taille maximale des accélérateurs est réduite. Dans le cas où les accélérateurs sont placés au niveau cluster, on obtient deux types de cluster : les clusters logiciels ($N \neq 0$ et $S = 0$) ; et les clusters matériels ($S \neq 0$ et $N = 0$). Les architectures ainsi obtenues sont appelées *HMpSoC partagés* (Figure 0-2b). Cette approche permet d'avoir des accélérateurs matériels de grande taille et de les partager entre les clusters. En contrepartie, ils induisent une augmentation des temps de communications entre la partie logicielle et la partie matérielle.

Les familles de HMpSoC introduites précédemment peuvent définir un grand nombre d'architectures. Pour cibler, avec précision, une architecture particulière au sein de ces familles, nous avons

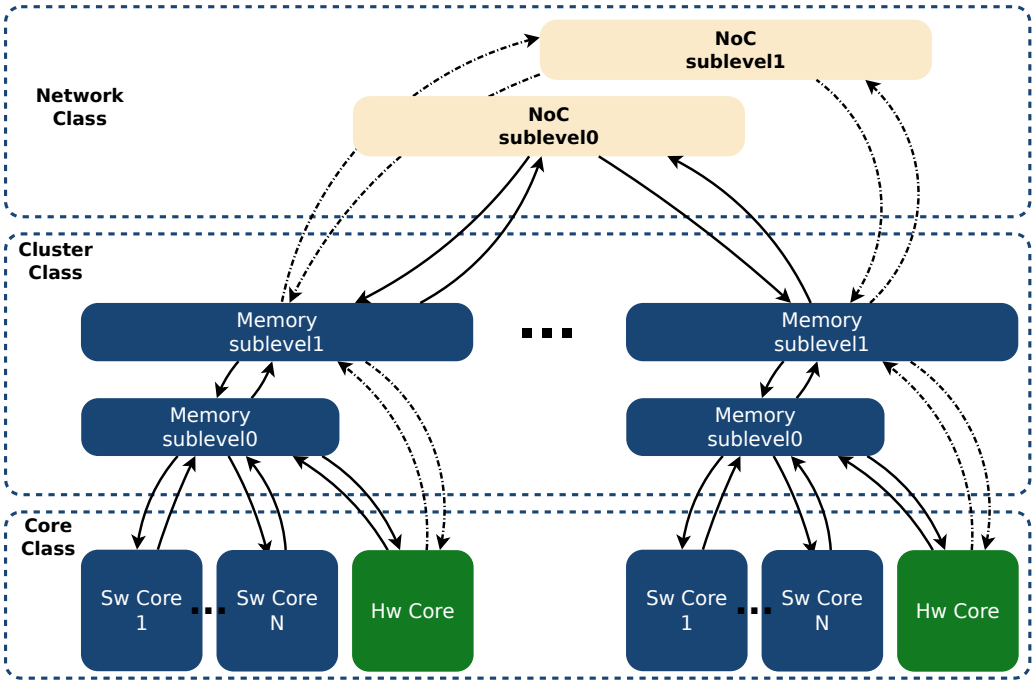


Figure 0-3 – **Représentation d'un HMpSoC basée sur la hiérarchie mémoire.**

besoin d'une représentation générique. Pour cela, nous avons proposé une représentation basée sur la hiérarchie mémoire décrite à l'aide d'un arbre orienté (Figure 0-3). Cette représentation divise la hiérarchie mémoire d'un HMpSoC en trois niveaux principaux : réseau, cluster et cœur. Chaque niveau contient des sous-niveaux, avec les propriétés suivantes.

- Au niveau réseau, les sous-niveaux sont disjoints de façon à ce que chaque sous-niveau puisse être utilisé indépendamment.
- Au niveau cluster, les sous-niveaux sont mixtes, c'est-à-dire qu'ils sont liés entre eux mais restent accessibles à chaque sous-niveau.
- Au niveau cœur, les sous-niveaux sont liés. L'accès au sous-niveau de profondeur L passe par les $L-1$ sous-niveaux supérieurs.

Pour décrire avec précision une architecture cible, les caractéristiques de chaque classe mémoire doivent être définies, telles que la profondeur de chaque sous-niveau, le débit des canaux, ainsi que le coût des communications.

2.2 Structure des applications

Après une étude des paradigmes de représentation des applications, nous formalisons la représentation qui sera utilisée dans le cadre de ce travail. Cette représentation permet d'exposer les parallélismes ainsi que les coûts de communications. Elle est basée sur un graphe à deux niveaux appelé *Energy Program Dependence Graph* (EPDG). Le niveau supérieur expose le parallélisme gros grain de l'application et contient des annotations tels que le coût d'exécution des nœuds et la taille des communications. Le niveau inférieur, quant à lui, expose le parallélisme à grain fin. Le niveau supérieur reprend la structure d'un graphe *Augmented Program Dependence Graph* (APDG) dans lequel les nœuds ne sont plus de simples instructions, mais des blocs d'instructions appelés *Basic power Bloc* (BpB). Ces BpBs représentent des macro-instructions avec un point d'entrée unique et un point de sortie unique et sont indépendants les uns des autres. Cela permet de les optimiser de manière

indépendante. De plus, ces blocs sont enrichis d'informations comme la taille des communications et le coût énergétique de calcul. Plus de détails sont donnés dans le chapitre 2.

3 Flot de conception & Contributions

Dans cette section, nous présentons les objectifs et les contributions principales de ces travaux. À cette fin, nous proposons une vue de haut niveau d'un flot de conception orienté énergie qui aidera les concepteurs à aborder les nouvelles architectures multicœurs hétérogènes. L'idée principale est d'intégrer un modèle de consommation rapide au plus tôt dans le flot de conception. Cela permet de considérer les performances énergétiques de l'application tout en explorant rapidement l'espace des possibles. L'approche proposée considère indépendamment l'impact des communications impliquées entre les organes de traitement et l'impact des calculs afin de proposer une formulation analytique qui diminue le temps d'estimation de la consommation énergétique.

La Figure 0-4 présente une vue d'ensemble du flot de conception, servant de fil conducteur aux travaux de cette thèse. Ce flot est divisé en quatre parties distinctes. La première consiste à extraire le parallélisme d'une application décrite de manière séquentielle et fournit un graphe de tâches parallèles. La seconde enrichit le graphe obtenu avec la taille des communications et identifie des blocs indépendants pour former des BpBs. Ces BpBs sont ensuite utilisés pour construire un graphe parallèle hiérarchique avec le formalisme EPDG. La troisième étape du flot de conception utilise les propriétés/caractéristiques de l'architecture cible pour enrichir l'EPDG avec le temps d'exécution et la consommation énergétique de chaque BpB. Enfin, l'espace de conception peut être exploré en prenant en compte l'énergie pour obtenir un front de Pareto des configurations en fonction de l'énergie et de la vitesse de calcul. Ces différentes étapes sont détaillées ci-après et reliées aux contributions de ces travaux.

3.1 Extraction du parallélisme

Cette première phase utilise une description séquentielle de l'application en entrée et repose sur les multiples outils proposés par la communauté pour extraire le parallélisme de l'application (voir chapitre 3 sous-section 2.1) et fournir un graphe de tâches parallèles décrivant l'application. Cette étape se base sur une approche itérative qui permet à l'utilisateur de modifier la structure du programme d'entrée pour améliorer les résultats d'extraction.

3.2 Creation de l'EPDG

Cette deuxième partie utilise en entrée le graphe de tâches parallèles précédemment obtenu ainsi qu'une description de l'architecture cible contenant le degré de parallélisme utilisable. Ces informations permettent de calculer la taille des communications entre chaque tâche du graphe. Les petites tâches sont ensuite fusionnées entre elles ainsi que celles impliquant un gros flux de communication dans des macro-tâches. Ces macro-tâches sont ensuite enrichies de deux nœuds supplémentaires représentant les communications entrantes (*CommIn*) et sortantes (*CommOut*). Des BpBs sont alors obtenus. Ils sont ensuite enrichis d'informations sur la taille des communications et reliés entre eux en fonction des dépendances pour obtenir un EPDG. À ce moment du développement, l'EPDG ne contient aucune information sur le coût énergétique de calcul ou sur le temps d'exécution.

3.3 Estimation de l'énergie

À ce stade, nous avons un EPDG sans information de coût et aucune idée des potentielles performances d'implémentation de chaque BpB. Cette étape repose également sur des outils extérieurs, tel que Aladdin [Sha+15], pour obtenir une estimation précise des performances atteignables par les implémentations des BpBs. Il est aussi envisageable que l'utilisateur puisse utiliser des outils de synthèse de haut niveau (*High Level Synthesis (HLS)*), ou une solution conçue à la main, pour obtenir ces valeurs de performances. Quelle que soit la méthode utilisée, les valeurs de performance obtenues en termes de consommation et de temps de calcul sont insérées dans le flot de conception pour enrichir le EPDG.

Lorsque tous les BpB possèdent des informations de performance, un outil d'estimation de puissance est utilisé. Cet outil d'estimation permet d'obtenir pour chaque BpB sa consommation énergétique ainsi que son temps d'exécution. Ces valeurs sont nécessaires pour chaque implémentation matérielle et pour chaque type de processeur logiciel disponible. Ces informations de coût sont insérées dans l'EPDG afin de pouvoir réaliser l'étape d'exploration.

3.4 Exploration de l'espace de conception

Les travaux de cette thèse se concentrent principalement sur cette partie. Son point de départ est un EPDG contenant les informations relatives à la taille des communications ainsi qu'au coût/temps d'exécution des BpBs pour chaque implémentation. Ces informations sont ensuite utilisées pour construire des configurations associant les implémentations utilisées pour les BpBs, le placement et l'ordonnancement des BpBs, etc. Pour chaque configuration obtenue dans l'espace de conception, le coût réel de consommation est estimé. Au vu du nombre de configurations à traiter, le temps de calcul est primordial. Pour répondre à cette problématique, un modèle de consommation analytique basé sur l'étude des communications est proposé dans le chapitre 4. Après avoir développé une infrastructure de mesure complète sur l'architecture Zynq de Xilinx, nous avons validé ce modèle de consommation sur une série d'applications générant différents schémas de communication.

Malgré le temps d'estimation très rapide, il reste impossible d'explorer l'ensemble de l'espace de conception de manière exhaustive. Pour pallier cette problématique, nous proposons, dans le chapitre 5, une méthode basée sur une formulation de programmation linéaire en nombre entier mixte (MILP). Cette dernière permet d'obtenir la configuration optimale pour des applications tuilées en moins d'une seconde. Pour des noyaux de calcul de type multiplication de matrice et filtre stencil, nous avons observé un gain de plus de $12\ \%$ sur la consommation par rapport à une approche traditionnelle.

L'objectif à plus long terme est de proposer des algorithmes d'exploration plus aboutis permettant d'obtenir un front de Pareto de configuration en fonction de la consommation énergétique et du temps de calcul. L'utilisateur pourra ainsi sélectionner la configuration idéale pour les contraintes de son système.

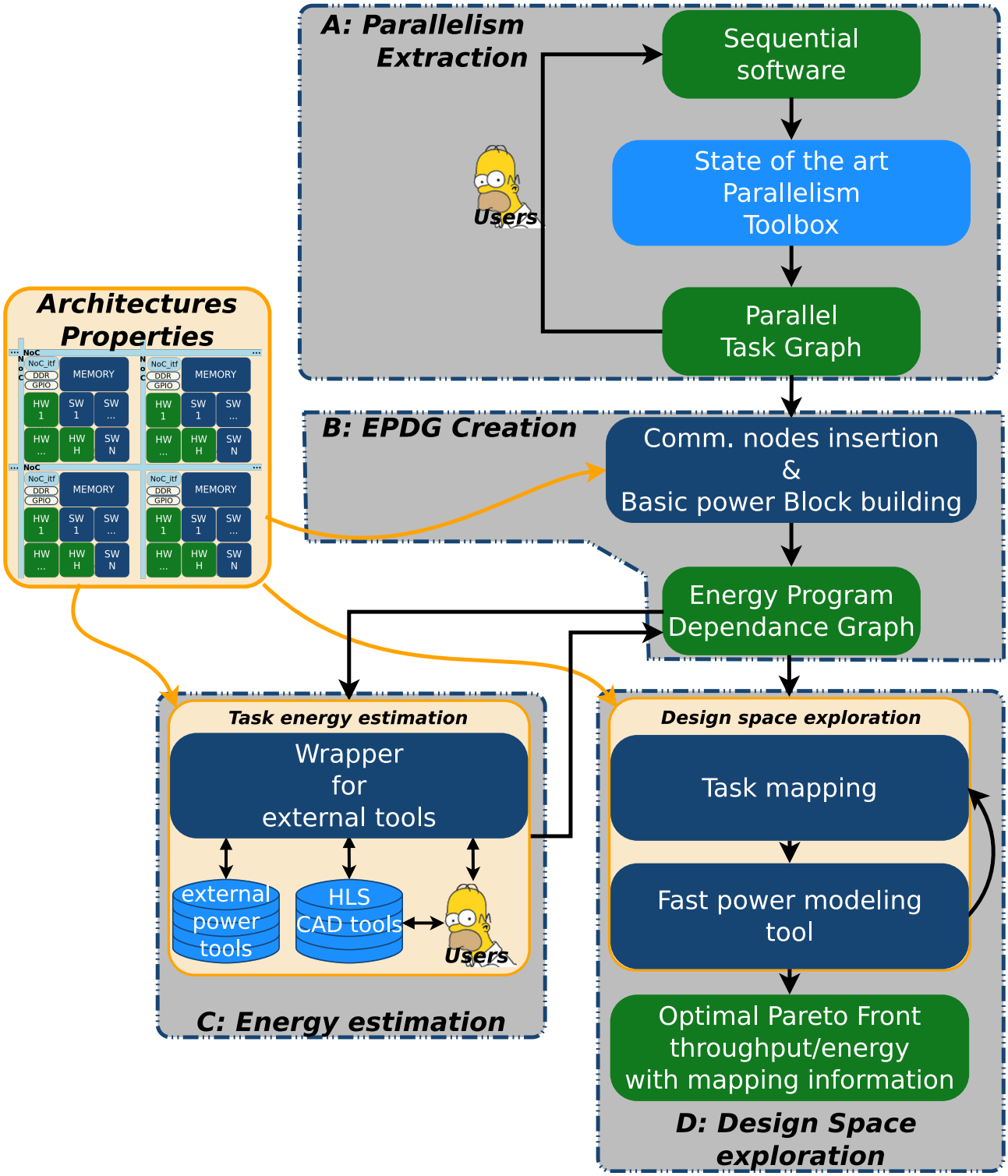


Figure 0-4 – **Vue d'ensemble du flot de conception orienté énergie.**

3.5 Platforme d'émulation de HMpSoC

Les différentes expérimentations menées pour l'élaboration du modèle rapide de consommation ainsi que la méthode d'exploration MILP ont mis en évidence différentes difficultés inhérentes au développement de nouvelles méthodes d'exploration. La plus récurente est de tester et valider les méthodes proposées sur un grand nombre d'architectures et d'applications distinctes. En effet, pour effectuer la validation sur une nouvelle architecture, il est nécessaire de développer un framework de mesure permettant la récupération des informations de consommation. Cela nécessite du temps et requiert la présence de capteurs sur la cible, ce qui n'est pas toujours le cas. Au niveau des applications, de nombreux jeux de benchmarks existent avec diverses caractéristiques, mais de la même manière le portage de ces applications sur l'architecture cible est très gourmand en temps.

Pour pallier à cela, nous proposons, dans le chapitre 6, une plateforme d'émulation d'architectures HMpSoC configurable qui permet de simuler un grand nombre d'architectures. Cette dernière associe les technologies QEMU et SystemC TLM-2.0 permettant d'atteindre de très bonnes performances en termes de temps de simulation. La structure interne des clusters utilisés dans la plateforme d'émulation est présentée dans la Figure 0-5.

De plus, cette plateforme d'émulation intègre un système d'estimation de l'énergie et un système de génération d'applications représentatives basé sur l'association aléatoire de divers noyaux de calcul. Cette architecture va permettre le test automatisé et à grande échelle d'algorithmes de partitionnement HW/SW et de placement de tâches sur des architectures HMpSoC, et ainsi faciliter l'adoption de ces nouvelles architectures.

4 Conclusion

Les travaux de cette thèse se concentrent sur l'élaboration d'un nouveau flot de développement efficace en énergie pour des applications de radio logicielle embarqués hautes performance ciblant des architectures multicœurs hétérogènes. Après un tour d'horizon des différentes architectures généralement utilisées dans ce domaine ainsi que des types de traitements impliqués, nous avons proposé une formalisation générique de la structure des architectures cibles ainsi que des applications. Sur ces bases, nous avons proposé un flot de développement ciblant l'efficacité énergétique et utilisé ce dernier comme fil conducteur lors de cette thèse. Pour répondre aux différentes problématiques inhérentes à son utilisation, nous avons introduit plusieurs méthodes et outils. Dans un premier temps, nous avons proposé un modèle de consommation rapide basé sur les communications qui permet d'adresser la phase de placement de tâche sur une architecture hétérogène. Ensuite, nous avons proposé une formalisation MILP permettant de trouver la configuration optimale en terme d'énergie pour le placement d'applications tuilées sur ce type d'architecture. Les expérimentations précédentes ont soulevé des problèmes annexes au flot de conception proposé avec notamment les difficultés liées au test des algorithmes d'exploration sur une large gamme d'architectures et d'applications. Nous avons donc introduit une plateforme d'émulation d'architectures HMpSoC embarquant un modèle de consommation ainsi qu'un générateur d'applications. Cette plateforme ouvre de nouvelles opportunités dans le domaine des algorithmes de partitionnement HW/SW et de placement des taches en permettant leur validation à grande échelle. Cette thèse aura donc permis de mettre en évidence les problématiques limitant l'adoption des architectures HMpSoC et propose des outils pour répondre à ces problèmes et faciliter les futures recherches.

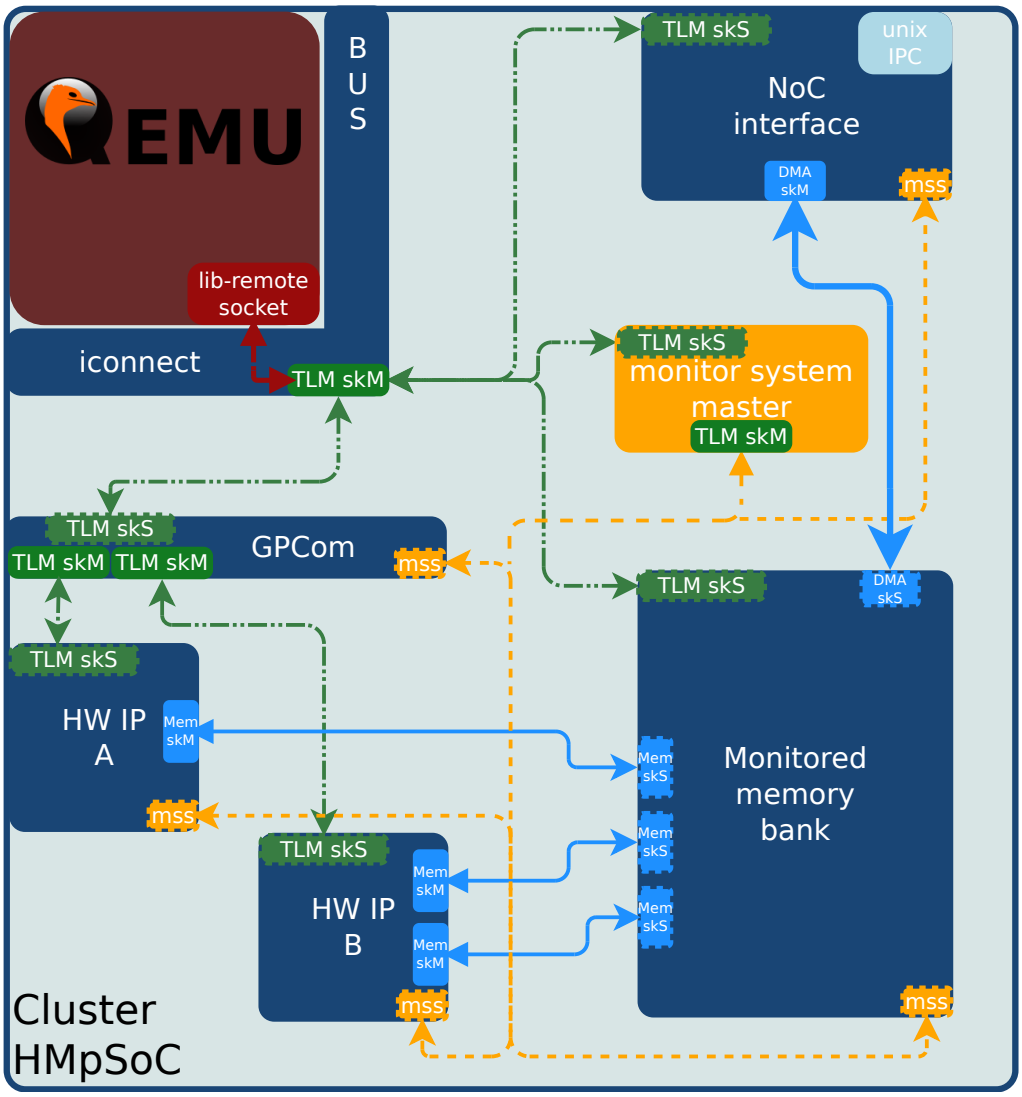


Figure 0-5 – **Structure des clusters de la plateforme d'émulation.**

CHAPTER **ONE**

INTRODUCTION

Contents

1 General context

Embedded systems have, per definition, limited energy resources. One of the major challenges of these systems is to integrate new functionalities within these energy constraints or under a maximum power dissipation constraint. One application that comes into this context is the implementation of the data link (mainly video stream) between drones and a base station or other drone features such as target detection and tracking. Indeed, energy management is another major challenge that limits the drone mission duration. Thus, reducing the power consumption of digital processing is a key for such embedded systems.

This thesis was co-founded by INRIA and the french ministry of defense (DGA). The starting context of this work is the DGA drone applications. These applications use a wireless data link communication based on the DVB-S2 standard. The current implementation of the communication link relies on three different kinds of architectures: *Field Programmable Gate Array (FPGA)*, *Digital Signal Processor (DSP)*, *Central Processing Unit (CPU)*. The use of three distinct chips hardens the reconfiguration process and decreases the overall power efficiency. The initial objectives of this thesis are to explore the opportunities provided by the new heterogeneous architectures, associating general purpose CPU with hardware accelerators, emerging on the market with these kinds of application and to slightly move the implementation to a new flexible paradigm based on Software-Defined Radio (SDR). Our objective is to find out the strengths and weaknesses of these architectures and to experience the difficulties that they involve within the development cycle of an energy-efficient implementation. The long term objective is to propose a generic target architecture and development solution that could be extended to a broader range of applications.

As a representative example of High-Performance Embedded Computing (HEPC), in the following, we detail the foundation principle of the SDR and we give an overview of the DVB-S2 standard.

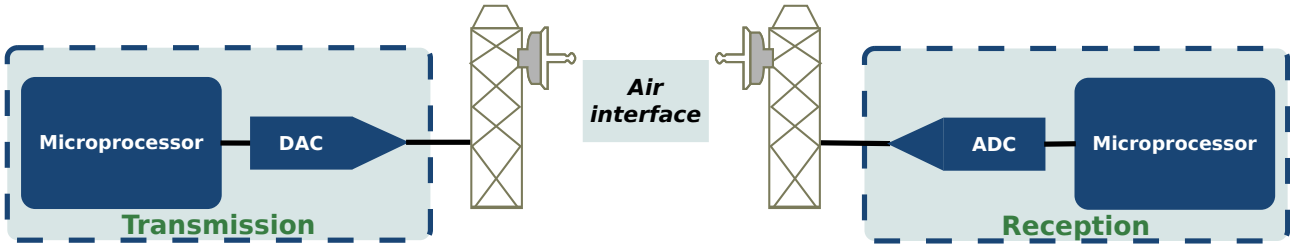


Figure 1-1 – **General structure of an ideal Software-Defined Radio (SDR).**

Then, we introduce the evolutions of semiconductor technologies that lead to new architectural trends.

1.1 Software-Defined Radio

The SDR principle was first introduced by Joseph Mitola in the 90s [Mit93]. SDR is the generic terminology that is employed to depict a flexible Digital Signal Processing architecture with very high reconfiguration capabilities so as to adapt itself to various air-interfaces. The SDR concept takes its roots within the military domain and the need of ensuring the inter-operability of the equipment through platforms that could run various types of waveforms and standards by a simple reprogramming or reconfiguration. Indeed, military applications have quite a long life cycle and it is crucial to maintain them and ensure or guarantee their inter-operability with the upcoming applications.

An ideal SDR structure can be illustrated as in Fig. 1-1. In this ideal representation, the signal is directly converted at the transmitter antenna and receiver antenna by a Digital to Analog Converter (DAC) and an Analog to Digital Converter (ADC), respectively. Such architecture delegates the computation requirements to programmable digital components such as microprocessors. SDR could support any type of waveforms since it is easily reprogrammable for a desired waveform implementation. The ideal SDR would have been the holy grail for digital radio systems, however, some limitations appear when it comes to practical requirements. Actually, digitizing the signal right after the antennas requires high sampling frequency ADC and DAC technologies capable to support the high-rate incoming data stream, which can represent a bottleneck.

In addition, microprocessor-based SDR implementations exhibit some limitations compared to hardware counterparts. These limitations are related to the power consumption and the achievable throughput. However, most of the DSP algorithms are highly parallel and require computation intensive solutions that could draw benefits of the new available heterogeneous many-core architectures and thus could tackle the power-consumption issues at the cost of a more complex programming.

In the next section, we present the structure of the second version of the Digital Video Broadcasting-Satellite (DVB-S2) wireless communication chain to expose the main processing and computations involved in this field of application.

1.2 DVB-S2: a relevant SDR application example

The DVB-S standard was created by the ETSI in 1995 and was the first standard developed in Europe. It relies on QPSK modulation and convolutional coding followed by an interleaver and a Reed-Solomon code. This structure enables high-quality transmissions even with low signal to

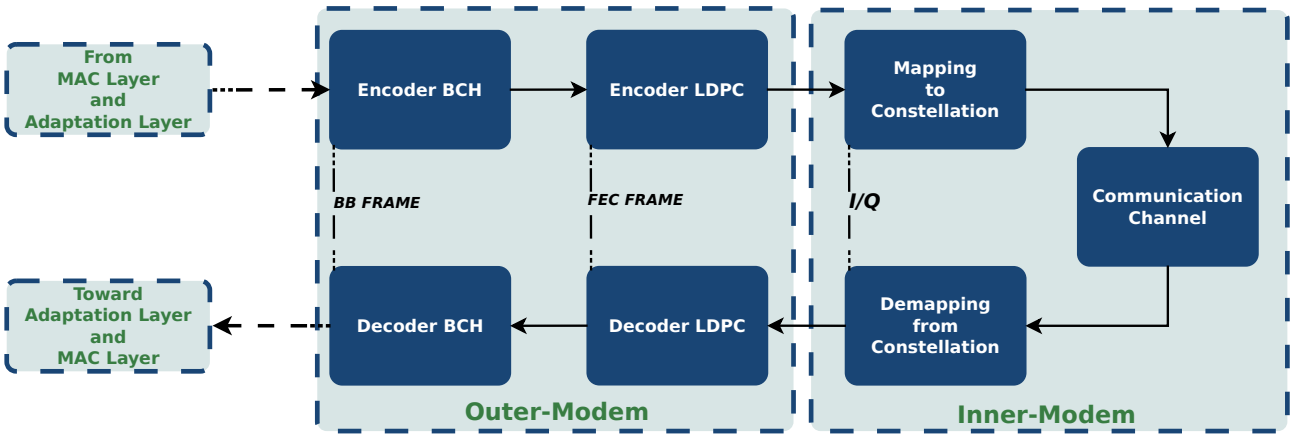


Figure 1-2 – **General structure of the DVB-S2 standard.**

noise ratio [97]. Finalized in 2004, the DVB-S2 [04] [05] standard was an evolution of the previous DVB-S standard. This new revision provides a better use of the spectral resources and a larger set of services through satellites. To improved the Quality of Services (QoS), DVB-S2 adopts an adaptive modulation and coding scheme. This scheme includes both Low Density Parity Check (LDPC) and Bose-Chaudhuri-Hocquenghem (BCH) channel coders with a near-to-1 coding ratio. Furthermore, DVB-S2 proposes a large panel of modulation schemes (QPSK, 8PSK, 16-APSK, 32-APSK). The choice of modulation and coding efficiency leads to a large range of spectral efficiency. In this section, we detail the structure and the computation requirements of the DVB-S2 transmission and reception chain.

The DVB-S2 standard is composed of a set of computation blocks (Fig. 1-2). Each block on the transmission line has its counterpart in the reception line. The computation involved in each block significantly differs and requires different hardware resources. In a general way, the transmitter involves less computation than the receiver.

BCH encoding: BCH codes form a large class of multiple random error-correcting binary codes. They were first discovered by A. Hocquenghem in 1959 and independently by R. C. Bose and D. K. Ray-Chaudhuri in 1960. BCH codes are cyclic codes, i.e. multilevel, cyclic and variable-length error correction codes. These codes enable an accurate control of the error encoding and correction capacity. The BCH encoding phase could be easily done with a low consumption custom hardware component based on Linear Feedback Shift Register (LFSR) [AS15]. The decoding phase is based on a more complex linear algebra method called syndrome decoding.

LDPC encoding: LDPC codes were first introduced in 1962 by R.G. Gallager [Gal62]. Due to the hardware limitations at that time, these codes could not actually be implemented. This was no longer true with the appearance of the iterative decoding process, and the LDPC were reintroduced in 1996 by D.J.C. Mackay [Mac99]. LDPC codes are linear block codes. They can be defined by a parity check matrix H of size $M \times N$. The number of columns N in H represents the code length and the number of rows M represents the number of parity check functions that the code needs to satisfy. The density of non-zero elements in the H matrix is very low, and this is why they are called low density parity check codes. The encoding phase involved a series of binary *Xor* ($\oplus$) operations applied on the transmission symbol with irregular patterns. LDPC decoding phase is based on the so-called message passing algorithms which are iterative algorithms such as the belief propagation algorithm. Their name derived from the fact that, at each round of the algorithm, messages are passed from message nodes to check nodes, and from check nodes back to message nodes. The messages from

message nodes to check nodes are computed based on the observed value of the message node and some messages passed from the neighboring check nodes to that message node. The LDPC decoding phase stays complex and requires a large amount of computation at each decoding iteration.

In a more general manner, the physical layer of radio transceivers could be divided into two main parts: the inner modem; and the outer modem. The inner modem includes environment parameter estimation, data detection, and transmission. The outer modem performs the encoding and decoding of frames from the received/transmitted data stream. On top of those compute-intensive layers, a Multiple Access Control (MAC) layer could be necessary to ensure the timing and acknowledgment schemes. Those elements involve different signal processing tasks with their own duty cycle. For example, the inner modem is mainly composed of irregular computation schemes strongly connected to the targeted standard. On the other side, the outer modem involves more regular computation schemes and requires less flexibility. Furthermore, those computation blocks expose different types of parallelism (task-level, instruction-level, and data-level parallelism). To reach a good energy/performance ratio, it is more efficient to use the task level to split the application across multiple computation units, in order to independently tune each of these computation units (cf. Chapter 2 Section 3).

2 Technology progress and architecture trends

For the past three decades, Moore's Law [Moo65] (roughly the doubling of on chip transistors every 18 months) has been a fundamental driver of the processor evolution. Coupled with the Dennard scaling [Den+74], a series of circuit, architecture and compiler advances has emerged and led in exponential performance increase. Since the beginning of the century, processor designers have shifted from single-core processor performance increase to core count increase to fully exploit Moore's Law scaling. Figure 1-3 depicts this phenomenon [BSW15]. The failure of Dennard scaling, partially addressed by the shift to Multi-processor System on Chip (MpSoC), may soon limit the multicore scaling just as single-core scaling has been curtailed. This leaves the community with no clear scaling path to exploit the still continuous transistor count increases. Lead by a new phenomenon called "Dark Silicon", the integration of dedicated heterogeneous accelerators within microarchitecture begin to appear.

2.1 Towards the integration of dedicated HW accelerators

The failure of the Dennard scaling exposed through the limitation of the supply voltage scaling led to the "Dark silicon" issue [Esm+11]. In fact, the current transistor density and the limitation of transistor voltage scaling prevent the use of all the available transistors at the same time due to the overall power that could be dissipated by a chip. This is the physical factor that led to "Dark Silicon". The percentage of a chip that can switch at full frequency drops exponentially with integration density and thus energy efficiency is not scaling along with technology advances. Furthermore, this limitation can also come from the algorithm structure. Indeed, the available degree of parallelism within an algorithm is limited and thus cannot efficiently use a 100-core or even more a 1000-core chip. This parallelism limitation leads to the same issue: all available cores within a chip could not be used at the same time. Therefore, it could be interesting to use part of the available transistors to efficiently implement a dedicated task. These transistors will not be used all the time, but clearly improve performance when used. This phenomenon is highlighted through the integration of spe-

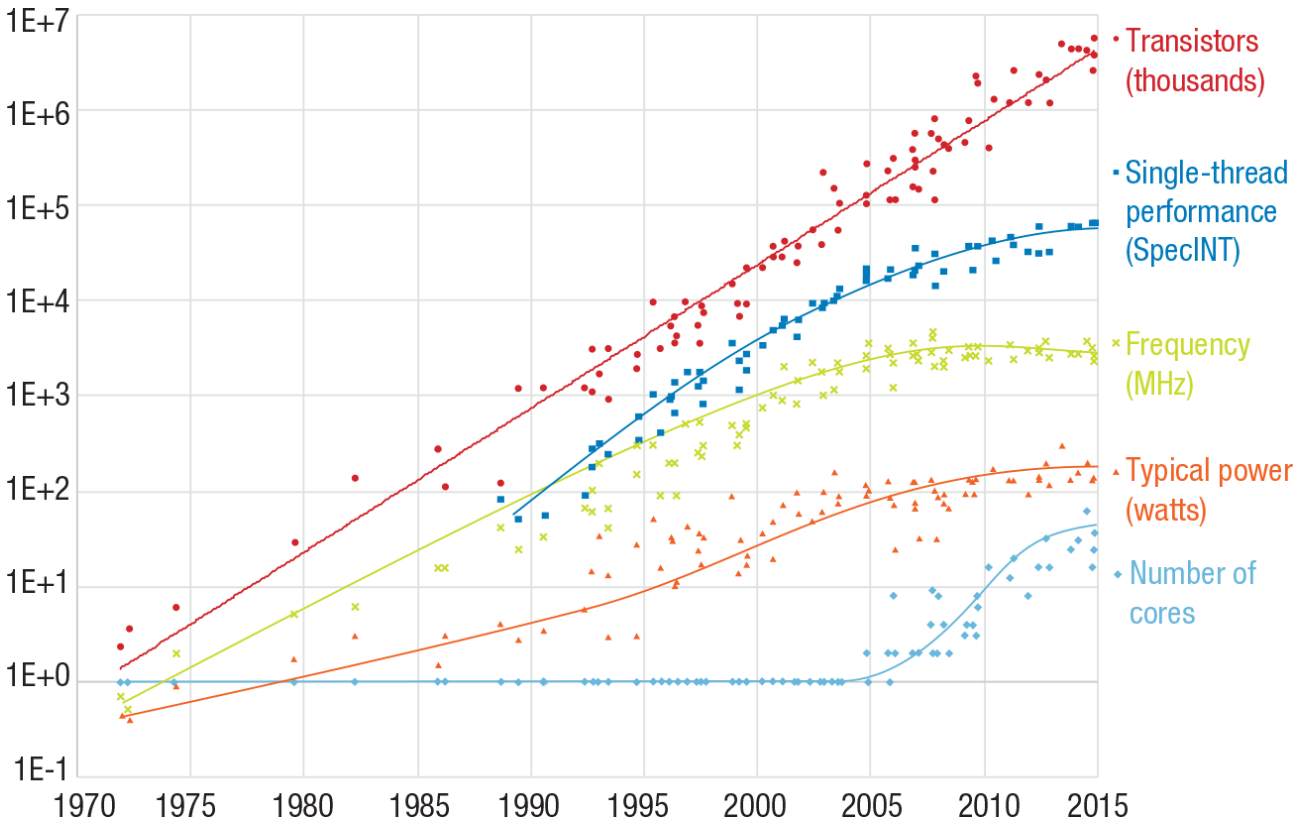


Figure 1-3 – **Processor microarchitecture and performance evolution [BSW15].**

cialized on chip HW accelerators such as Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA). This is the rise of Heterogeneous MpSoC (HMpSoC).

2.2 Energy efficiency: the next challenge of embedded systems

On top of this continuous research of performance improvement, a new key factor emerged: energy consumption. Indeed, with the democratization of embedded and wearable systems, that run on a limited power budget, energy consumption that directly impacts their autonomy is becoming a commercial issue. With their dedicated HW accelerators, the new class of HMpSoC architectures could address the challenge of energy efficiency. However, they introduce complex and hard-to-solve issues on SW/HW partitioning and task mapping. The high number of available cores and implementation types leads to a huge design space. Consequently, when energy consumption is a key requirement of the application, this solution space must be explored, early in the design phase, to obtain efficient solution. For this purpose, the overall design methodology needs to be revised to integrate fast and accurate power estimation tools. In the next section, we detail the different contributions that aim at addressing these objectives.

3 Contributions of the Thesis

This thesis proposes new methodologies and tools to address the complex task of energy-aware task mapping on HMpSoC architectures. For this purpose, we start by studying the properties and characteristics of different architecture structures that target the SDR field of application. Then, we look at the structure of multiple general purpose many-core architectures as well as heterogeneous

architectures. These observations have allowed us to propose a generic definition of the target architecture family of this work alongside with a representation method.

Then, we propose a fast power modelling methodology for the HMpSoC architectures. This model focuses on the communications involved by the task mapping across the computational cores of the target architecture. We propose and validate an extraction method to use this power modelling methodology on real architectures.

This model is then used within a Design Space Exploration (DSE) method, based on a Mixed Integer-Linear Programming (MILP) formulation, which computes the best task mapping of tiled applications over a HMpSoC. The DSE method is coupled with an extraction method of the application/architecture characteristics that allows its use with other applications and architectures.

This modelling work highlighted the difficulties linked to the design and test of new energy-aware DSE. With this in mind, we present a HMpSoC emulation platform that embeds power modeling features and a representative test application framework. The obtained emulation platform enables the designer to simulate the behavior of application execution on a broad range of HMpSoC architectures in a reasonable time and open new perspectives for the design and test of energy aware DSE. This emulation platform was fully developed during this thesis.

4 Organization of the document

This thesis is organized as follows. The first part introduces the context of the work and proposes some related work pertaining to this thesis. Chapter 2 presents an overview of some dedicated SDR architectures alongside with general purpose many-core architectures. Thanks to these observations, we define the generic HMpSoC structure used in the following alongside with a representation method based on the communication channels. We also expose widespread application representation paradigms and precisely define the one used in the thesis work.

Chapter 3 presents a detailed state-of-the-art on multiple aspects of the work conducted during this thesis. Power modelling methods are presented at different levels of abstraction. Available methods and tools for parallelism extraction are also detailed. The chapter ends with the introduction of current DSE methods.

The second part of the thesis details the contributions of this work. In Chapter 4, a communication-based power modelling tool is thoroughly presented. A description of the model structure is given alongside with a set of methods to use this model with real architectures. The methods and models are then used on a real architecture to validate this work.

This power modelling method is then used in Chapter 5 within a DSE tool that enables the mapping of tiled applications on a HMpSoC. This DSE tool proposes an energy-driven accelerator exploration for HMpSoC and enables the designer to select the optimal association of SW and HW computation block with the best processing distribution between the blocks. This method is validated on two application kernels on the Xilinx Zynq architecture.

Chapter 6 introduces an energy-aware HMpSoC emulation platform that enables to simulate the execution of a broad range of representative applications on various HMpSoC architectures. This chapter depicts the implementation structure of the emulation platform and the configuration facilities. The execution models used and the corresponding execution framework are also depicted. Then, the method of application generation is presented as well as the automated application execution mechanism through mapping directives.

Finally, Chapter 7 summarizes the contributions of this thesis and outlines the perspectives on future works.

Chapter **TWO**

HETEROGENEOUS ARCHITECTURES: STRUCTURES AND APPLICATIONS

Contents

In the first section of this chapter, some architectures dedicated to SDR are depicted. A highlight is given on the heterogeneous part introduced to answer the computation challenges involved by the communication chain. In addition, we present general purpose MpSoC with potential heterogeneous features and show how we could draw performance from them in the SDR context. Then, we propose in Section 2 a generic definition of heterogeneous many-core architecture alongside with a generic description model used in the following works. In Section 3 we introduce structures and representations employed to model applications. Finally in Section 4, we conclude with the representation of the application that we use in this work.

1 Survey on MpSoC architectures

The introduction highlighted that the use of heterogeneous multi-core architectures can enhance the power efficiency of the solution. In the following, we detail how heterogeneity was introduced within MpSoC architectures. We particularly focus on architectures dedicated to SDR applications. Then, we present some general-purpose architectures and how SDR applications could draw benefits from them.

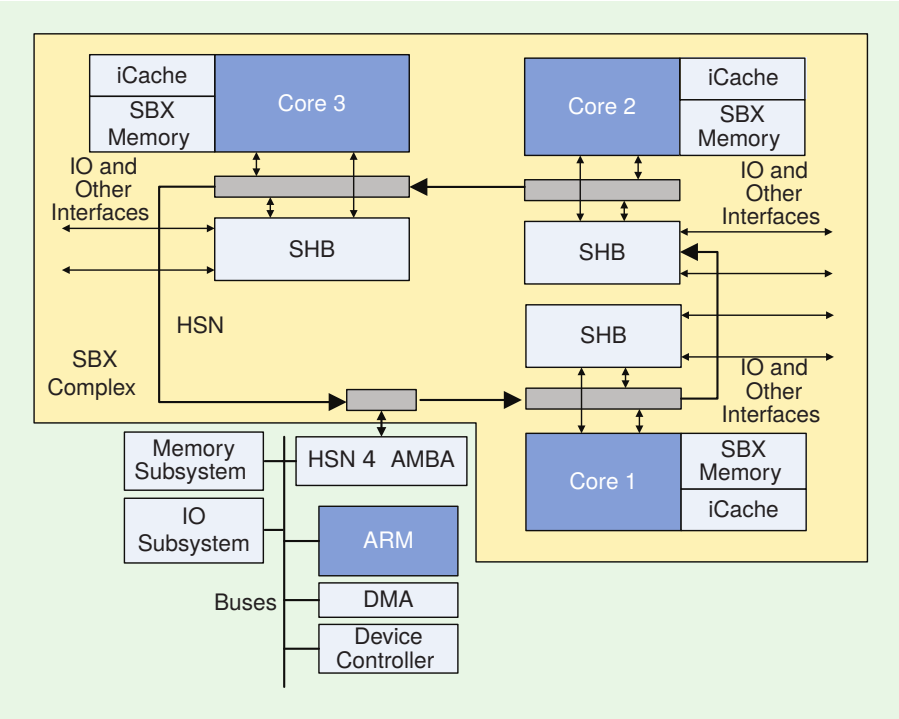


Figure 2-1 – **Sandbridge SB3500 platform architecture [Pal+10].**

1.1 Dedicated SDR architectures

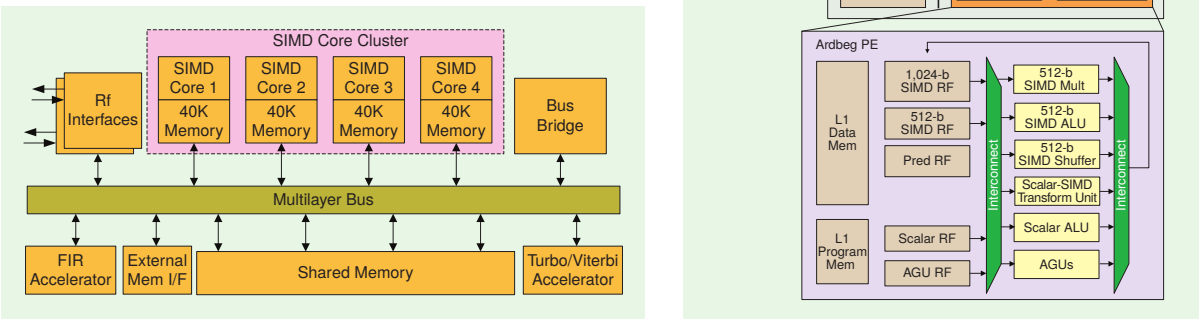
This section presents an overview of the available SDR architectures that embed both HW and SW cores. Their general structures are depicted with a focus on their heterogeneity. More details could be found in the survey on SDR platforms proposed by Palkovic *et al.* [Pal+10].

Sandbridge SB3500 [Nac08]: The SDR platform SB3500 from Sandbridge is composed of four processor cores. The control and platform management tasks are performed by one ARM processor (Fig. 2-1. The inner- and outer-modem computation tasks are performed by three custom SIMD sandblaster cores developed by Sanbbridge. This platform proposes a limited amount of heterogeneity through the use of two kinds of processor architecture.

Infineon Music [Ram07]: The Music platform (Fig.2-2a) from Infineon gathers four SIMD cores with various accelerators such as FIR filter and turbo/viterbi accelerators. The SIMD cores coupled with the filter accelerators are dedicated to the inner-modem computation. The outer-modem treatments are implemented mainly within the turbo/viterbi accelerators.

ARDBEG Platform [Woh+08]: A collaboration between ARM and the University of Michigan led to the ARDBEG platform (Fig.2-2b). This platform is composed of three SIMD processors, one dedicated to control and the two others dedicated to inner-modem treatments. The outer-modem computation is performed with the help of a hardware turbo accelerators.

IMEC'S BEAR Platform [Bou+08]: The Base band Engine for Adaptive Radio (BEAR), developed by IMEC, is a heterogeneous multi-core architecture (Fig.2-3) composed of six processor cores coupled with two hardware accelerators. The six processors consist of one general purpose ARM processor dedicated to control and MAC layer, three Application Specific Instruction Processors



(a) **Infineon MuSIC-1 platform architecture [Pal+10]**

(b) **ARM/University of Michigan's Ardbeg architecture [Pal+10]**

Figure 2-2 – **Architectures with dedicated FEC accelerators.**

(ASIP) dedicated to coarse time synchronization (DFE processing block), and two architectures for dynamically reconfigurable embedded systems (ADRES) dedicated to inner modem computation. The ASIPs are built around a 5-way VLIW with two scalar and three vector computation lines. The ADRES cores are highly flexible and energy efficient Coarse Grain Reconfigurable Architectures (CGRA). Finally, two hardware accelerators dedicated to Viterbi decoding are added. All those elements are connected through Advanced Micro-controller Bus Architecture (AMBA).

The four architectures presented consist of mainly two approaches. On the one hand, architectures such as Sanbbridge SB3500 (Fig. 2-2a) focus on reconfigurability at a cost of a low heterogeneity with no use of dedicated hardware accelerators. This approach enables to address a wide range of communication standard at the cost of energy-efficiency and performance. Platform such as Infineon Music and ARDBEG are more mitigated (Fig. 2-2). They use the heterogeneity by using dedicated FEC hardware accelerators, which improves performance and energy-efficiency but narrows the spectrum of targeted standard. On the other hand, the BEAR platform (Fig. 2-3) focuses on heterogeneity with a large panel of dedicated accelerators such as domain-specific reconfigurable accelerators and dedicated FEC accelerators. The obtained performance per watt is high, but the use of dedicated FEC components introduces the same issues as before. Furthermore, if we focus on the aspect of cost and time-to-market, the development of a new architecture for each new kind of communication standard is not really efficient.

1.2 General Purpose architectures

In this section, we briefly present some general-purpose MpSoC architectures. As those architectures are not necessarily heterogeneous, we proposed some solutions or suggestions to introduce reconfigurable hardware components in them.

Network on Chip With the scaling of microchip technologies, the number of heterogeneous computation units integrated within a single chip has exploded. The communications between those elements encounter fundamental physical limitations such as time-of-flight of electrical signals, power

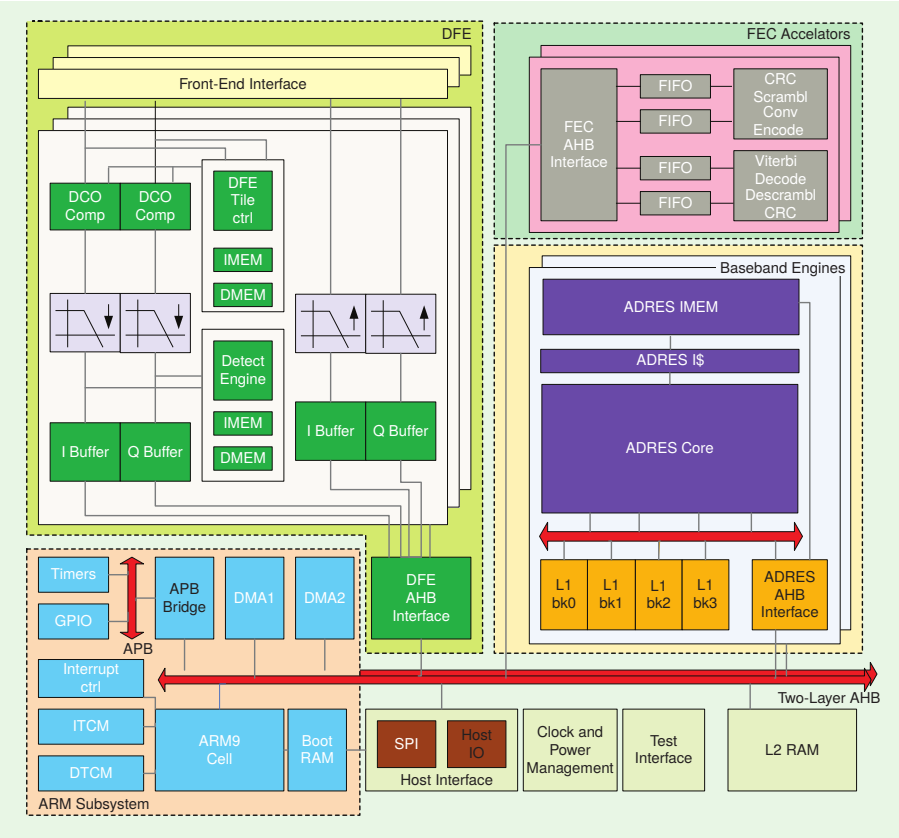
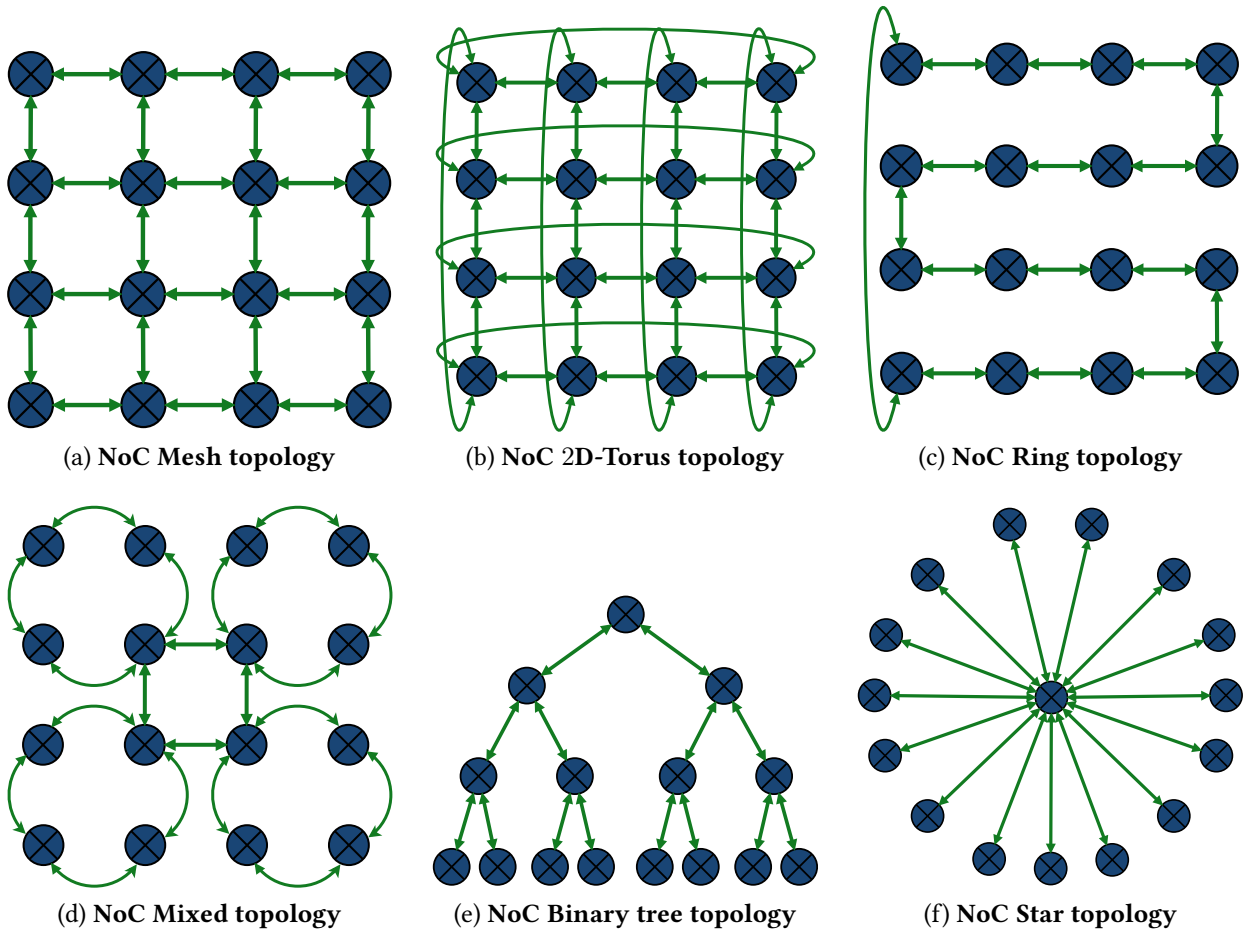


Figure 2-3 – **IMEC's BEAR SDR platform [Pal+10].**

use in driving long wires/cables, etc. The well understood concepts of bus communications introduce numerous drawbacks that prevent the scaling of these technologies within the many-era. In fact, every new units added on a communication bus introduce a parasite capacitance that degrades the electrical performance of the wires. Once the communication between two entity was established the latency is wire-speed, nevertheless the bandwidth is limited and shared between all the units. Then, with the multiplication of the master units on the bus, the arbitration delay grows and becomes a bottleneck. To overcome theses issues and get a scalable communication infrastructure, the concept of Network on Chip (NoC) was introduced. With this approach, for all network sizes, only point-to-point one-way wires are used, thus local performance is not degraded when scaling. The routing decision could be well distributed if the network protocol used is non-central. The overall aggregated bandwidth scales with the network size, nonetheless the internal network contention may cause latency and degrade the performance.

The NoC latency depends on the physical characteristics of the hardware implementations, the distance between two units expressed in number of hopes within the network and the network contention. The two last elements are strongly connected with the application communication pattern and with the network topology. Fig. 2-4 shows a panel of the most widespread NoC topologies. The used topology influences the average number of communication hopes and the network congestion. For example, Fig. 2-4c shows a ring topology which is simple to manage and arbitrate. As a counterpart, this topology presents a high number of average hopes and it is very sensitive to congestion. The mesh topology (cf. Fig. 2-4a) lowers the average number of hopes but the traffic could be easily saturated in the middle of the NoC. The 2D-torus topology solves this issue, but requires more area on the chip. The choice of the target topology is hard and its efficiency is tightly linked to the application communication pattern.



(a) **NoC Mesh topology** (b) **NoC 2D-Torus topology** (c) **NoC Ring topology**

(d) **NoC Mixed topology** (e) **NoC Binary tree topology** (f) **NoC Star topology**

Figure 2-4 – **Panel of common NoC topology.**

Kalray MPPA: Kalray's Massively Parallel Processor Array (MPPA) architecture [Din+13] is a homogeneous MpSoC which is mainly composed of 256 VLIW processors gathered into 16 clusters. All clusters are linked together through two NoCs, one with low bandwidth for control information and one with high-bandwidth for data communications. External communications are managed through four I/O clusters. Figure 2-5 shows a simplified block diagram of a MPPA-256 processor chip. Each cluster contains 16 5-way VLIW processors and NoC interface. On each side of the cluster array, an I/O cluster provides access to external Double Data Rate (DDR) memory banks and to PCIe and Ethernet interfaces. The use of power-efficient VLIW computational cores alongside with the homogeneous and widely parallel architecture enables an efficient 50 GFLOPS/Watts ratio for the first version and forecasts to meet a ratio up to 100 for the third one. Furthermore, the design of the embedded NoC enables its access directly through the IO as an external cluster. This opens the way for heterogeneous computing by connecting an external reconfigurable accelerator such as FPGA that can directly communicate with the NoC.

Mellanox TILE-Gx: The TILE-Gx architecture [Mel] includes an array from 36 up to 72 tiles (cores). Each tile is built around a 64-bit 3-way VLIW processor, integrating three levels of cache, and a non-blocking switch that integrates the tile into a power-efficient interconnect mesh. The TILE-Gx architecture offers 23.5 MB of on-chip cache with the Mellanox's (formerly Tilera) dynamic distributed cache technology that provides a 2× average improvement in cache coherence performance over traditional cache coherence protocol. Each tile can independently run a complete OS or multiple tiles can be grouped together to run a multi-processing OS like Symmetric Multi-Processor (SMP) Linux. A dynamic power management unit enables the tiles to be put into a low-power sleep

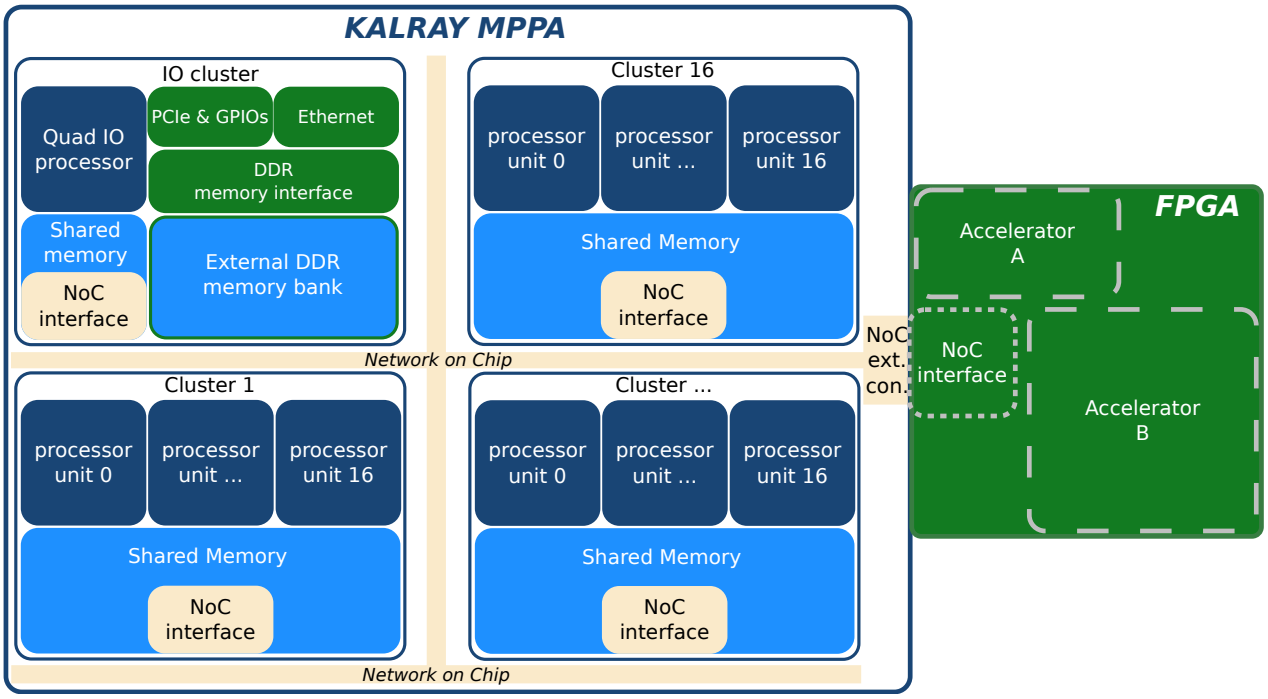


Figure 2-5 – **Kalray's MPPA architecture extended to heterogeneous.**

mode independently and achieved a ratio close to 25% GFLOPS/Watts. This architecture can be extended by connecting multiple chips through a shared DDR memory or Ethernet interfaces. Including hardware accelerators, such as some FPGA, establishes heterogeneous architecture.

Xilinx Zynq: The Zynq architecture from Xilinx [Cro+14] is representative of heterogeneous targets as it combines two ARM cortex A9 processors with an FPGA fabric (see Fig. 2-7). The Zynq computation cores communicate through different memory levels: L2 cache and DDR. Each level can be accessed from two channels: one for SW and one for HW. Furthermore, synchronization and configuration communications can occur from a dedicated channel without memory bank access. This enables low-latency communications with zero-copy approach. The low-power processor cores and the dedicated reconfigurable accelerators coupled with the low-latency communication scheme enable to address a wide range of embedded applications with good energy-efficiency.

As shown in Sub-Section 2.1 the actual architectural trend and the new dark silicon paradigm led to the multiplication of the computational cores and to the integration of heterogeneous accelerators. The NoC communication scheme, that is now used inside SoC, enables the scalability of the architecture and leads to a new paradigm of core simplification combined with their multiplication, which enhances the computational power per Watts. This section presents multiple approaches to take advantage of these elements. The first option was to start building dedicated architecture but this solution reduces the scope of the produced chips and increased their cost. The second solution was to add reconfigurable accelerators inside general purpose architectures. The reconfigurability of the accelerators coupled with the computational power of the high number of cores and the communication flexibility given by the NoC enable to target a wide number of applications with efficiency on the same architecture. This approach opens up wide opportunities at the cost of hardening the application development. In fact, with those highly parallel and configurable architectures, the slightest choice on application implementation could have a strong effect on energy consumption and performance. The design space of those architectures is wide, but its full exploration is more essential than ever.

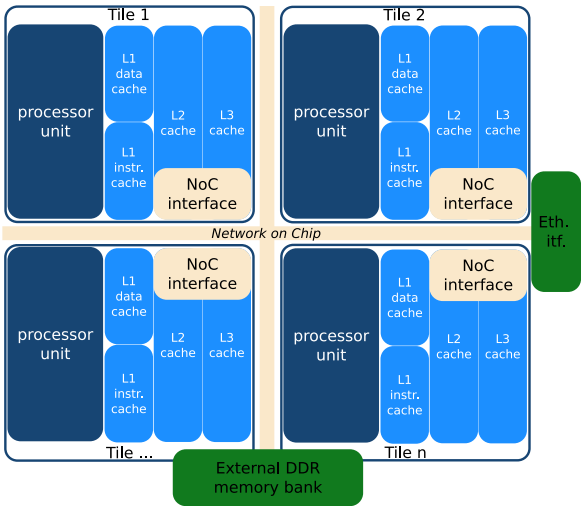


Figure 2-6 – **Mellanox's Tile-Gx architecture.**

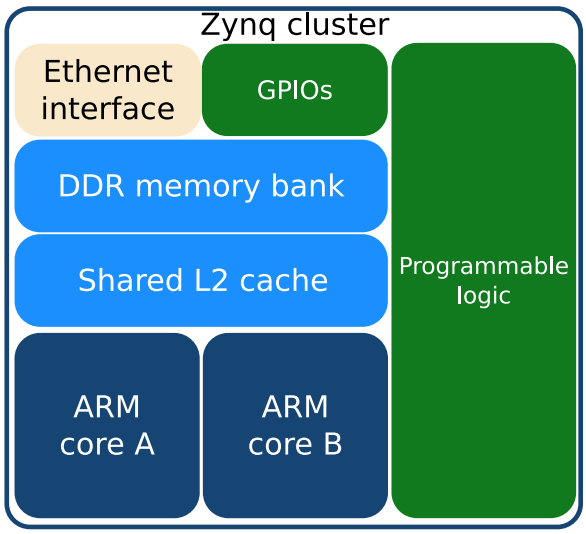
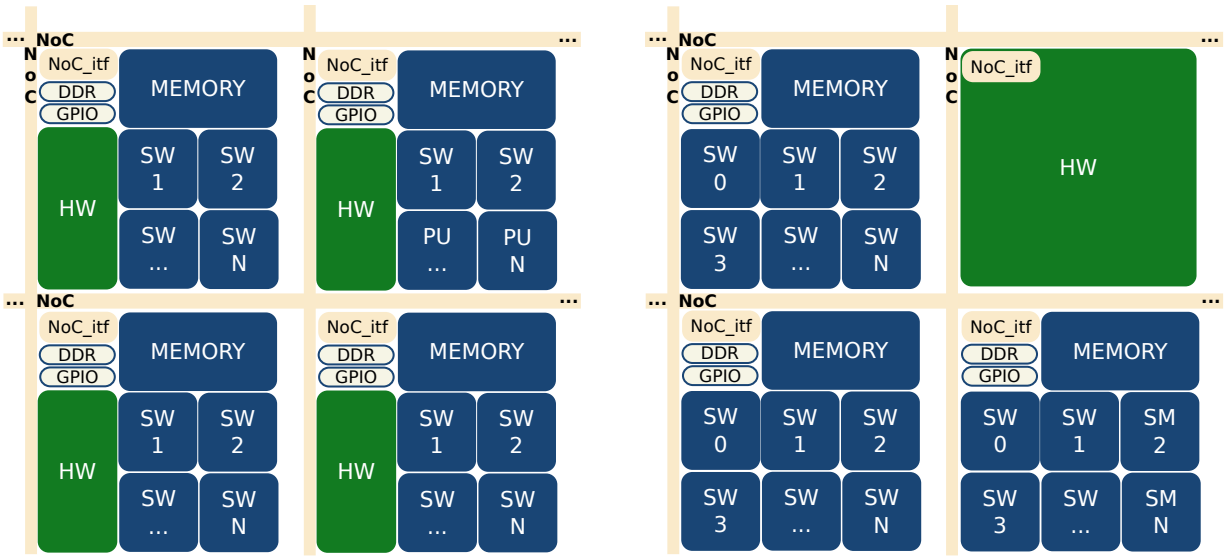


Figure 2-7 – **Xilinx's zynq architecture.**



(a) **Distributed HMpSoC.**

(b) **Shared HMpSoC.**

Figure 2-8 – **HMpSoC architectures.**

2 Generic HMpSoC representation

This Section introduces the target architecture families of this work. First we present in Sub-Section 2.1 the HMpSoC definition and subfamily characteristics. In Sub-Section 2.2, we propose a generic representation of these architectures with a focus of the internal memory hierarchy and on the available communication channels.

2.1 Generic HMpSoC architecture

Sub-Section 1.2 has shown a set of representative and commercially available many-core, potentially heterogeneous, architectures. In this section, we summarize their general properties and build a formal definition of the HMPSoC. As shown before, MpSoCs are generally composed of a set of memories, processors, interconnecting elements and I/O peripherals. When associated with

specialized hardware accelerators, MpSoC are referred to as heterogeneous, in the sense that they combine software (SW) processors with hardware (HW) accelerators. A generic representation of a HMpSoC is formalized and shown in Fig. 2-8. This HMpSoC architecture is built around clusters linked together through NoCs. Each cluster is composed of up to N SW cores coupled with HW accelerators of size S. S could represent the surface available within an ASIC or the available resources in an FPGA. At the cluster level, the communications occur through shared memory banks. From this description, different families of HMpSoC can be built depending on the mapping of the integrated HW area in the architecture. When the HW area is placed at the processor level in each cluster ($S \neq 0$), *Distributed HMpSoCs* (Fig. 2-8a) are obtained. They enable fast communications between the SW and HW parts. Consequently, the maximum area of a hardware accelerator is reduced. When the HW area is placed at the cluster level, two cluster types are involved: the SW one with $N \neq 0$ and $S = 0$ and the HW one with $N = 0$ and $S \neq 0$. HW area is therefore shared between SW clusters. These *Shared HMpSoCs* (Fig. 2-8b) induce an increase in communication time and latency between the SW and HW parts. On the other hand, the total HW size could increase and sharing the accelerators between SW clusters becomes possible. As communications mainly rely on the memory hierarchy of the architecture, the memory hierarchy will be particularly reported. The NoC used within those architectures could be built around the state-of-the-art topologies presented in Sub-Section 1.2.

The heterogeneous MPPA extension depicted in Fig. 2-5 belongs to the *Shared HMpSoCs* family. As the opposite, the extension of the Zynq architecture to HMpSoCs leads to the *Distributed HMpSoCs* family. Such an extension could be easily implemented with a cluster of Zynq circuits linked together through Ethernet interfaces as shown on Fig. 2-9.

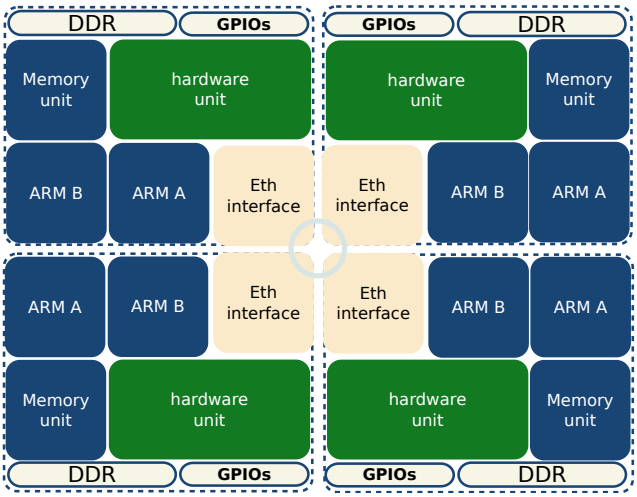


Figure 2-9 – **Cluster of Zynq architectures.**

2.2 Oriented tree for memory hierarchies

The HMpSoC architectural properties introduced in the previous section can be used to define a large number of architectures. To target a particular architecture within those families, we need a generic representation. For this purpose, we introduce in [Rou+16] the oriented-tree memory hierarchy representation depicted in Fig. 2-10. The memory hierarchy of a HMpSoC can be classified in three main levels: network, cluster, and core. A representation of this memory hierarchy as an oriented tree, equivalent to Fig. 2-8a, is shown in Fig. 2-10 (the generic memory hierarchy representation can also be applied to Fig. 2-8b). Each level contains sublevels, with the following properties:

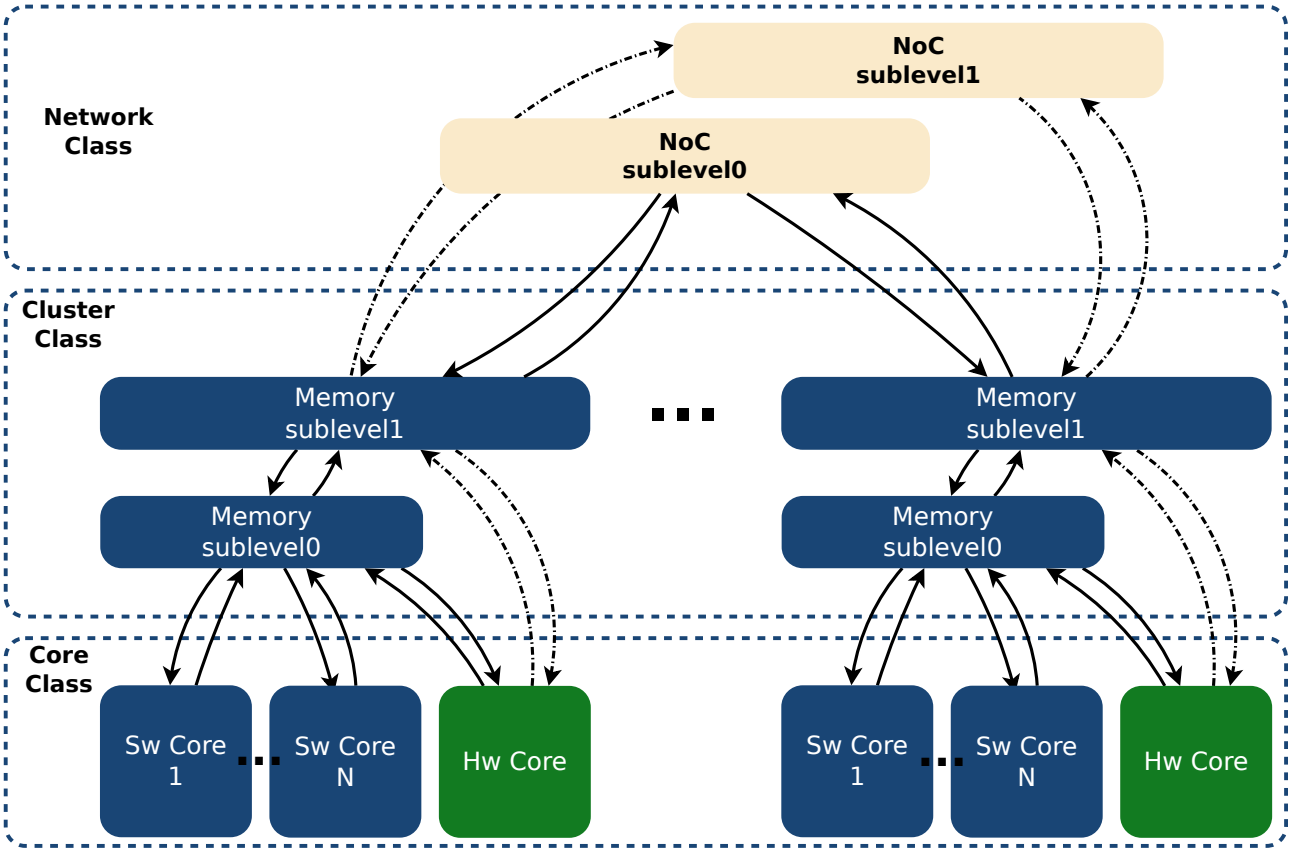


Figure 2-10 – **Oriented-tree memory hierarchies abstraction.**

- At the network level, sublevels are separated such that each sublevel can be independently used.
- At the cluster level, sublevels are mixed such that sublevels are chained but can be accessed at each sublevel.
- At the core level, sublevels are chained. Accessing sublevel at depth L crosses the $L-1$ upper sublevels.

To describe precisely a target architecture, the characteristics of each memory class have to be defined, such as the depth of each sublevel, the channel throughput, and so on.

3 Application structure and representation

This section introduces the different structures of application parallelism that could be used to exploit the parallel structure of the heterogeneous architectures. Then, a panel of application representation paradigm is presented in order to show how application structure could be defined and used as a starting point for a DSE.

3.1 Parallelism-Level

In order to get the full performance from MpSoC architectures, the target application needs to be executed in parallel. The application is split in different parts and spread across multiple processors or computation cores and executed simultaneously in parallel. With the advent of heterogeneous

architectures, those parts could be executed at different rates and on different computation structures. As a result, if part of the application produces/consumes data read or produced by another part, no guarantee can be made about the rate of execution and a particular attention should be paid on block interaction, data dependency, and synchronization. The parallelization task is a two-fold step. First, it must recognize when different regions of an application may produce, or consume, data produced by other regions. Second, it should transform the application structure to reduce the amount of interaction between regions that can prevent parallelization.

Also, the application parallelism could be exploited at different granularities. The choice of the granularity is strongly connected to the structure of the target architectures. Some of the main parallelism categories are detailed below. Parallelism extraction algorithms and tools are available and will be detailed in Sub-Section 2.1.

Instruction-Parallel: This is the finest parallelism degree. It is based on the dependency analysis between instructions. When two instructions are independent, they could be executed in parallel. This parallelism degree is dynamically exploited in Out-Of-Order architectures with the help of dependency analyses at run time. In VLIW architectures, this level of parallelism is statically exploited by the compiler.

Data-Parallel: This type of parallelism is defined by the fact that the same set of operations could be independently applied to multiple blocks of data. The parallelism degree is intrinsically defined by the data division granularity.

Loop-Parallel: For many applications, loop iterations can be independently executed in parallel. For this purpose, the iteration space may have to be transformed with techniques such as loop-skewing [JMF01]. The parallelism degree is defined by the division applied on the loop iteration space.

Task-Parallel: An application could be represented as a set of tasks. Tasks are portion of program that may be simultaneously and atomically executed. The number of available tasks is generally wider than the available computation cores so that tasks need to be mapped/scheduled at the appropriate moment to prevent dependency violations. A task-parallel application could be represented with a task graph showing multiple tasks and irregular communications between them. This approach can usually represent an equivalent form of both the data-parallel and loop-parallel paradigms by defining loop iterations and portions of the data as the individual tasks.

Pipeline-Parallel: The pipeline parallelism is usually used with many embedded applications written with a streaming-oriented structure. The pipeline approach divides loop body into disjunctive stages that are executed in overlapping/pipeline manner on multiple computation cores. This parallelism could also be directly exploited inside the computation structure of hardware accelerators.

3.2 Data Flow programming models

Programming models are characterized by the way in which the instruction executions are initiated and by the data communications involved between the instructions. In Data Flow models, when an instruction generates a result, the output data are directly passed to the next instruction

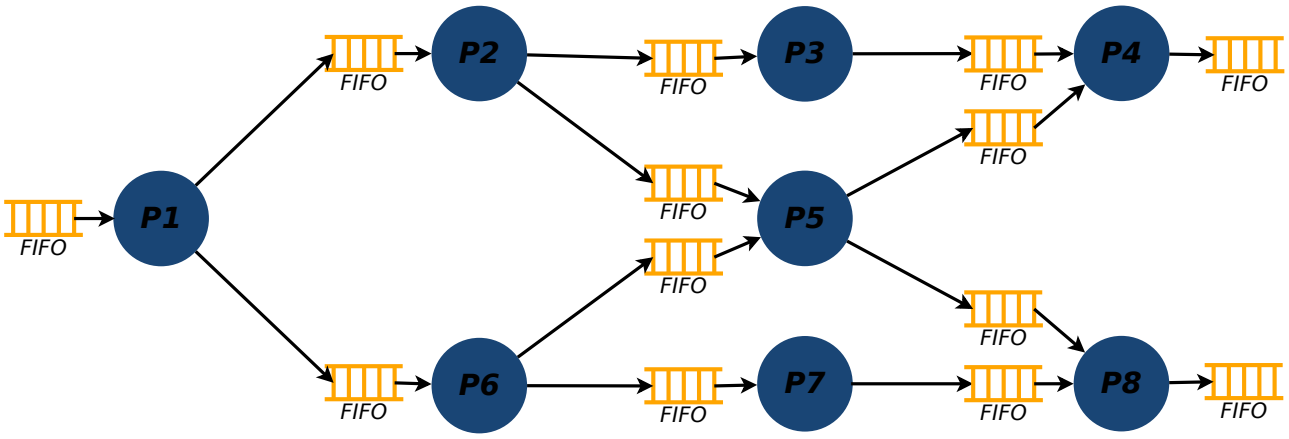


Figure 2-11 – **A sample KPN model.**

that consumes them. When multiple instructions consume the same result, separate copies of the data are created and dispatched to each instruction. The instruction executions are initiated when the complete set of inputs is available. Then, the instruction consumes the set and the data are deleted. The data flow programming model does not use the concept of shared named data container. Therefore, it is really suited for functional programming paradigm but the lack of updatable memory makes array data structure difficult to represent. This application representation has been declined in multiple forms over the years. Two representative ones are detailed below.

Kahn process network: The Kahn Process Network (KPN) [Kah74; KM77] is a software paradigm that splits an application in a network of concurrent processes that communicate in a point-to-point fashion with the help of unbounded FIFO channels. The synchronization between these processes is done through a *blocking-read* FIFO's primitive. Each process is a sequential program that could be autonomously executed, and concurrently with others. Kahn process networks make task-level parallelism and communication explicit and present the following characteristics:

- KPN is deterministic. Its outputs are independent of the evaluation schedule of the process network. This gives a large freedom of schedule when mapping a KPN on a real architecture.
- Simple synchronization through FIFO *blocking-read* mechanism could be easily implemented in software or in hardware.
- Process runs autonomously and the control is perfectly distributed over the network. No global manager is thus needed.
- The communications are only done through FIFO. There is therefore no notion of shared memory with concurrent access and race condition issues.

A sample structure of KPN is depicted on Fig. 2-11. This structure makes KPN really popular for describing the behavior of data flow and streaming applications such as audio, video and multimedia applications. However, the unbounded nature of the FIFO channels, the non-blocking write operations and the expressive power of KPN introduce some drawbacks. The main issue is that KPN cannot be statically analyzed. A run-time system is required to schedule the execution of the processes and to manage memory usage of the communication channels.

Data Flow Graph: The Data Flow Graph (DFG) is a more restrictive version of the process network that tries to circumvent the previous issue of static analysis. DFG introduces a new concept of firing rules that trigger the execution of each process, called now actors. The firing rules fix

the number of input data tokens and output data slots needed to start the execution of the actor. When the firing rules are satisfied, the actor fires. The firing rules enable to statically determine the schedulability of the network and the size of the communication channels needed. DFG has different subfamilies, each introducing its own restrictions on the firing rules, such as Synchronous Data Flow Graph (SDFG) or Cyclo-Static Data Flow Graph (CSDFG) [PPL95]. For example, in a CSDFG, the firing rules could be expressed with a static cycle. An internal state variable is added into each actor. Then the number of tokens produced and consumed is computed following the state of this variable and the firing rule cycle. The ΣC language [Gou+11] is an implementation of the CSDFG paradigm and was included into the Kalray's MPPA toolchain. The drawbacks of this approach come when we mapped on a real architecture an application in which the parallelism degree is greater than the available computation cores in the architecture. In fact, in that case, there are more agents to schedule than the available cores. To circumvent this issue, a dynamic task scheduler can be used. However, this introduces extra computations and unpredictable communications between the computation cores.

3.3 Control Flow programming models

In Control Flow models, the partial result generated by an instruction is indirectly shared through shared named data container. Once stored, the partial result could be used an unspecified number of time. The execution of an instruction is no longer initiated by the data availability but by a complete set of control signals. With the use of shared named container, Control Flow facilitates the representation of array data structure. This data mechanism relies on shared memory cell and incurs extra writes and reads that could decrease the memory performance such as frequency access. Control Flow models suite well with imperative languages and are easy to map on conventional hardware architecture such as *von Neumann* and *Harvard* architectures. However, it presents a real inability to represent and utilize parallelism. The most used Control Flow models is the Control Flow Graph (CFG) which is detailed below.

Control Flow Graph: The CFG is the most used representation inside compilers. In a CFG, the program statements are organized into basic blocks. A basic block is composed of a list of statements with exactly one entry point and one exit point. So that, a basic block could not contain a destination jump instruction, and only the last statements could start the execution of another basic block. A CFG is a directed graph composed of nodes that are the basic blocks of a program and two additional nodes, Entry and Exit. There is an edge from Entry to any basic block at which the program can be entered, and there is an edge to Exit from any basic block that can exit the program. The other edges of the graph represent transfers of control between the basic blocks.

3.4 Mixed programming models

Considering independently data or control dependencies limits the potential expressiveness of parallelism. To mitigate those limitations, some mixed representations, associating control and data dependencies, have been introduced. Some of them are briefly detailed in this section.

Control-Data Flow Graph: The Control Data Flow Graph (CDFG) is composed of basic blocks linked together with data and control dependencies. The CDFG exposes the parallelism at basic block level. Groups of instructions belonging to different basic blocks are sequentialized, while instructions belonging to the same basic block can be simultaneously executed, if they are data

independent. In [Tou11], a CDFG construction algorithm was proposed. It is based around two extraction modules, one dedicated to control information, the other to data information. The control flow section constructs a global CFG of the application including call stacks, loop nest trees, and normalized loop iteration vectors. Then, the data flow section is responsible for mapping memory addresses to specific high-level data flow information.

Program Dependence Graph: The Program Dependence Graph (PDG) [FOW87] makes both the data and control dependencies explicit for each instructions within a program. In PDG representation, the statements are represented as nodes, and the edges between nodes represent the control conditions and data values on which the operations depend. The PDG associates the data relationships and the essential control relationships, without the unnecessary sequencing presented in the control flow graph. PDG representation is useful for automatic detection and management of parallelism [Sar93] and for solving a variety of problems, such as optimization, vectorization, VLIW code generation. The PDG can also be extended with various cost information to obtain a so-called Augmented Program Dependence Graph (APDG) [Cor14]. An APDG is a regular PDG combining control and data flow dependencies within the same graph. Each node is augmented with the iteration count and execution costs of the represented statement. Each edge is enriched with the communication costs, the communicated data and the iteration count.

Hierarchical Task Graph: The Hierarchical Task Graph (HTG) is a program representation, proposed by Girkar *et al.* [GP94], that encapsulates minimal data and control dependencies. This representation can be useful for the extraction and exploitation task level parallelism. A HTG is a layered graph, in which each layer is a direct acyclic graph associating control and data dependencies. The HTG ensures that the graph at each level has a single entry and exit point, so there is a path from the entry node to every node, and from every node to the exit node. HTG is built with the help of two node types:

- Simple nodes: match with a basic statement in the original source code.
- Hierarchical nodes: match with loop or function bodies in the original code, the hierarchical nodes contain an arbitrary number of child nodes. These child nodes can be simple nodes or other hierarchical nodes.

This representation can be enhanced with performance and communication characteristics, which leads to the Augmented Hierarchical Task Graph (AHTG) [Cor14]. This new representation adds two node categories to encapsulate the communications between two distinct hierarchical levels of the graph, which are the in- and out-node communications. These communication nodes are crossed by the total communication flow within a hierarchical node and enable to independently extract the parallelism for each node.

As show in this section, numerous application representation paradigms exist. Some of them such as Data Flow approaches are well fitted to intrinsically expose parallelism, but the majority of legacy applications are not written with these paradigms and the state-of-art parallelism extraction tools (cf. Sub-Section 2.1) do not generate Data Flow graph as output. The HTG representation could also be interesting because the n levels of hierarchy enable to explore the parallelism granularity and to find the best one for a given architecture. However, the HTG extraction method proposed in [GP94] is not able to extract all the application parallelism. On the other hand, the APDG structure could be easily generated by extraction parallelism tools. Moreover, it gives a good overview of the data dependencies and can be enriched with communication and energy cost information. This is why we propose in the following section a twofold level application representation based on the APDG

structure that is convenient for task-level parallelism representation and suits well with the energy aware task mapping problematic.

4 Generic application structure

In this section, we formalize an application representation that enables to expose parallelism alongside with energy cost and communication information. These elements define the general context of this work. The methods and tools proposed in the second part of this thesis use as inputs the generic representations of both HMpSoC architectures (cf. Section 2) and applications as defined below. In some case, the proposed tools target a more restrictive context that will be discussed later.

In this work, we will consider that applications are described with a two-level graph representation called *Energy Program Dependences Graph* (EPDG). In this graph, the outer level representation exposes the task- and pipeline-level parallelism with annotations such as execution cost of nodes and communication size of edges. On the other hand, the inner level aims to expose the instruction-, loop- and data-parallelism. Within the EPDG, the outer level is built around the APDG structure where the node do not share statements anymore, but Basic power Block (BpB). BpBs are macro statements with single entry point and single exit point, that encapsulate sub-program and are designed to ease the task-based energy modelling. The BpB is built upon one of the previous representation depending of the underlying parallelism extraction tool used. Within BpB, the read access is drawn at the beginning of the block and write access pushed at this end through two extra nodes called respectively *InComm* and *OutComm*. These extra nodes enhance the sub-graph independencies and enable to independently optimize the BpB. In addition, the BpB is enriched with information such as communication size and computation energy cost.

The energy-aware design space exploration will be focused on the outer APDG and will add information to the graph such as node placement and communication channel used in the extra information. Fig. 2-12 displays the general structure of the EPDG representation.

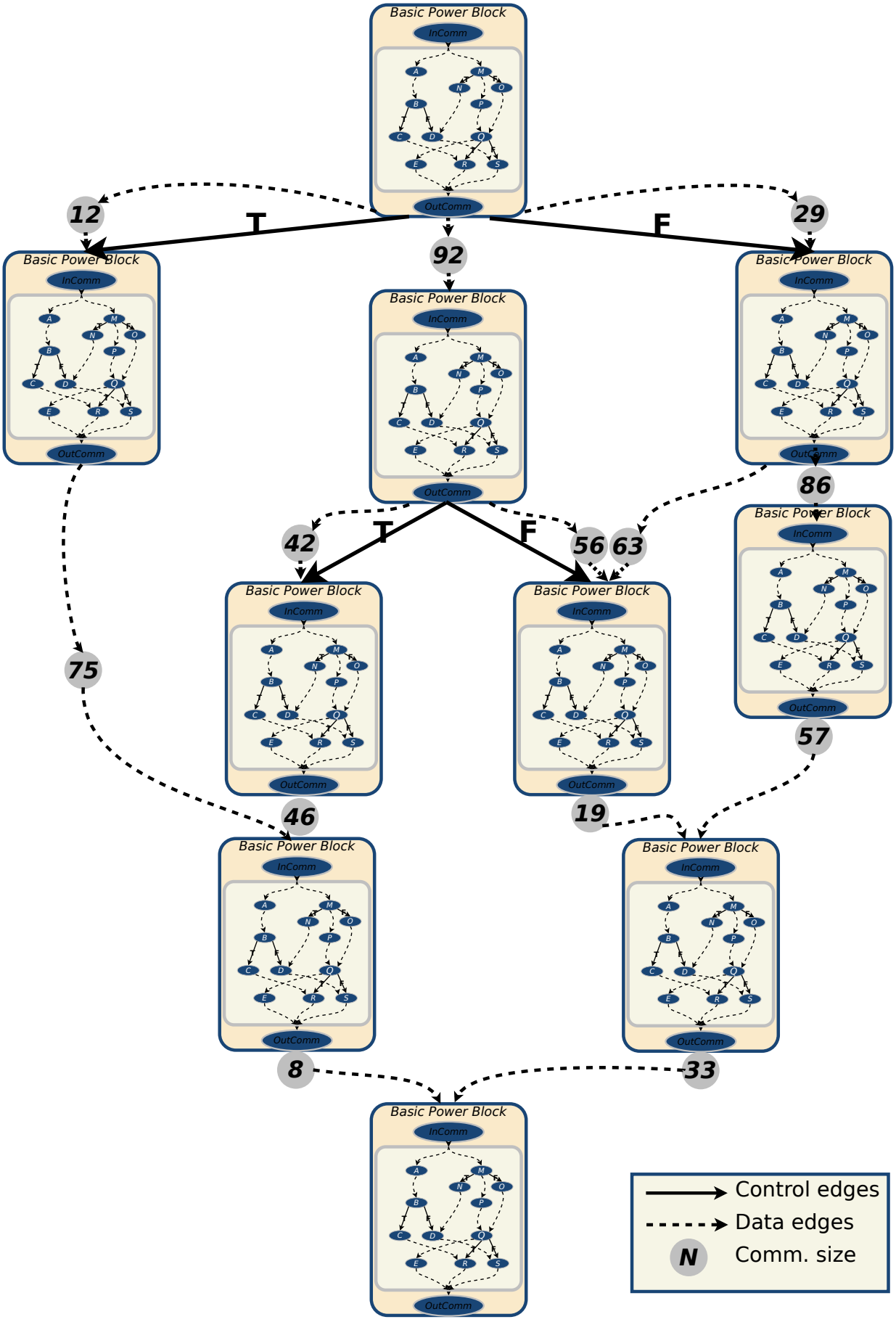


Figure 2-12 – **Overview of EPDG representation.**

Chapter **THREE**

POWER MODELLING AND COMPUTER-AIDED DESIGN TOOLS

Contents

With the advent of embedded systems, reducing energy consumption has become a key objective in the design of embedded applications. Indeed, due to increasing integration densities and clock frequencies, and the emergence of the "dark silicon" issue [Esm+11], the target architecture for embedded systems is moving towards massive multi-cores and heterogeneous computing systems. For these heterogeneous architectures, performance and energy consumption depend on a large set of parameters such as the HW/SW partitioning, the type of HW implementations or the communication costs. Design Space Exploration (DSE) consists in adjusting these parameters while monitoring a set of metrics (execution time, power, energy efficiency) to find the best mapping of the application on the target architecture. The size of the design space increases exponentially with design parameters, which highlights the need for fast exploration tools. Furthermore, with this new architectural trend, the number of design parameters involved in the energy consumption is significantly increasing. Therefore, an early estimation of the power consumption in the design flow becomes key.

In this chapter, Section 1 presents the state-of-the-art power modelling tools and discusses their usability within DSE. Section 2 focuses on CAD tools. Algorithms and tools available for extracting parallelism from sequential application are introduced in Sub-Section 2.1. Finally, DSE exploration tools targeting MpSoC and HW accelerator exploration are presented in Sub-Section 2.2.

1 Power modelling facilities

In a circuit, most energy is consumed during logic node transitions. This consumption source is called dynamic, while a second source, which corresponds to the quiescent state of the circuit, is called static. In the past, the static energy consumption could be neglected. However, with new nanometric technologies, the leakage current has increased and this is no longer true. In general, the power consumption of Complementary Metal Oxide Semi-conductor (CMOS) circuits is defined

as

$$P = P_{static} + P_{dynamic}. \tag{3-1}$$

Therefore, estimating consumption in SoC requires analysis tools capable of evaluating different static and dynamic sources when running the application. This analysis can be done in a more or less fine way, depending on the level of abstraction at which our system is described. In the literature, there are a variety of tools for estimating consumption at different levels of abstraction. These tools offer different compromises between the accuracy of the estimate and computation complexity which influences its speed. Some of them are presented hereafter.

1.1 Low-level estimation techniques

The efficiency of power estimation techniques is characterized by the speed of the simulator and the estimation accuracy. We present here some *state-of-the-art* low-level power modelling techniques which are frequently used. Low-level power estimation techniques could be based on a wide range of abstraction levels:

- Circuit/Transistor Level;
- Logic/Gate Level;
- Register-Transfer Level (RTL).

1.1-1 Circuit/Transistor level

With this approach the target architecture is represented with an extremely complex set of transistors and nets with the routing and layout information. This information is not delivered in the component data-sheet, which makes the circuit-level approach almost unfeasible in practice in our case. Moreover, the circuit-level simulation kernel uses component models based on linear differential equations solved in the continuous-time domain, which require a large amount of time even for small systems.

Spice: SPICE [Ber] is a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, lossless and lossy transmission lines, switches, uniform distributed RC lines, and the five most common semiconductor devices: diodes, BJTs, JFETs, MESFETs, and MOSFETs.

PowerMill: PowerMill [Hua+95], is a transistor-level simulator that enables the simulation of current and power behavior in Very-Large-Scale Integration (VLSI) circuits. It introduces a new transistor modelling technology and a versatile event-driven simulation algorithm. PowerMill is capable of simulating detailed current behavior in modern deep-submicron CMOS circuits, including sophisticated circuits such as exclusive-or gates and sense-ampliers.

Other tools exists such as Lsim Power Analyst from Mentor Graphics [Gra]. The estimated values obtained at this level are very close to the real circuit values. Unfortunately the simulation time is very important and constitutes a major obstacle to use these tools. In the case of MPSoC integrating hundreds of millions of transistors, the use of this approach to estimate the total consumption becomes impossible.

1.1-2 Logic/Gate level

Contrary to the previous methodology, the gate-level power estimation is based on an event-driven simulation kernel that works in discrete-time domain. This approach significantly reduces the estimation complexity with a near insignificant lost of accuracy [Bra00].

Chou *et al.* [CR96] introduce power estimation tool based on a Monte Carlo solving method that takes the logic signal spatial and temporal correlations into account, coupled with an accurate estimation of the signal activities at the sequential logic node level. The accuracy obtained with the Monte Carlo approach is within 2%, at the cost of a still long simulation time.

Other commercially available tools such as PowerGate from Synopsys [Syn] and DIESEL from Philips [DT01] exist and achieve the same kind of performance.

1.1-3 Register Transfer level

RTL is an abstraction level that models synchronous digital circuits as a set of registers and logical operations linked together by a flow of digital signals. Registers are implemented as D flip-flops and the logical operations are performed with blocks of combinational logic composed of logic gates. Most of the RTL power estimation tools derive the power properties of combinational blocks from an analysis, under defined conditions, of the block isolated from the rest of the design. Thus, the power consumption of a block is tightly coupled with the input statistics.

Bogliolo *et al.* [ALG00] have proposed a general methodology for building and tuning RTL power models. This methodology addresses both hard macros (presynthesized functional blocks) and soft macros (functional units for which only a synthesizable HDL description is provided). It exploits linear regression and non parametric extensions to express the dependency of power dissipation on input and output activity. It explains in details the bottom-up off-line characterization of regression-based power macro models. This approach introduces a low overhead on-line characterization method for enhancing the accuracy of off-line characterization.

Potlapally *et al.* [Pot+01] present a technique in which they do cycle-accurate power macro modelling of the RTL component. This technique is based on the fact that RTL components exhibit different power behavior for different input scenarios. They create power macro model for each of these behaviors, also known as power modes. Their framework chooses the appropriate power mode from the input trace in each cycle and then applies power macro-modelling techniques introduced in [ALG00] to get a power estimation. This technique is limited to the typical average power estimation scenarios and covers also non-trivial scenarios. However, the estimation speed is very slow.

1.2 High-level estimation techniques

The methods discussed previously achieve high accuracy but the simulation time is an obstacle. To circumvent this issue, researchers propose solutions to abstract the description of some implementation details. At the cost of a certain loss of accuracy, the estimation of the power consumption can be performed by considering events at higher granularity levels than the switching in a transistor or the change of state of a logic gate. Events actually identify activities which consume a

significant amount of energy in the modelling level considered. The energy consumption estimation is obtained in two steps. The first one consists in obtaining the occurrences of each relevant activity during an application execution. In the second one, the previous values are injected into the power models to calculate the consumption of each component of the system. To implement this methodology, several approaches, offering different trade-offs between accuracy and speed, exist and will be discussed in the next paragraphs.

1.2-1 Architectural-level

Architectural-level power modelling tools describe the target architecture on a component basis. The target architecture is split in elemental units called components. For the example of a processor, component could be the register file, the branch predictor, the instruction caches, the Translation Lookaside Buffer (TLB), etc. Next, some of the most relevant architectural-level power models are presented.

SimplePower: Ye *et al.* introduce the SimplePower framework [Ye+00] that enables the evaluation of the effect of high-level algorithmic and architectural trade-offs on energy. SimplePower estimates the energy dissipated in system memories and buses with an analytical model. The other power consumption sources are modelled with a cycle-accurate RTL that uses transition sensitive energy models.

Wattch: Wattch [BTM00] is a widely-used processor power estimation tool. It calculates dynamic power dissipation from switching events obtained from an architectural simulation and capacitance models of components of the micro-architecture. When modelling out-of-order processors, Wattch uses the synthetic Register Unit Update (RUU) model that is tightly coupled to the SimpleScalar simulator [ALE02] to emulate the Instruction Set Architecture (ISA). Wattch has enabled the computer architecture research community to explore power-efficient design options. However, limitations of Wattch have become apparent. First, Wattch models power without considering timing and area. Second, Wattch only models dynamic power consumption. Third, Wattch uses simple linear scaling models based on $0.8\mu m$ technology that are inaccurate to make predictions for current and future deep sub-micron technology nodes.

McPAT: McPAT [Li+09] is a power modelling framework that addresses the Wattch weaknesses and integrates a power, area, and timing modelling infrastructure. This approach solves a key issue in current tools: modelling power and/or area without considering timing constraints. McPAT supports all important components needed to model modern multicore/manycore architectures with in-order and out-of-order processors. McPAT considers three types of power dissipation: dynamic, static, and short-circuit. This enables McPAT to accurately handle recent deep sub-micron technologies in which static power has become comparable to dynamic power.

Avalanche: The Avalanche framework, proposed by Henkel *et al.* [LH98; Hen99] introduces an energy consumption estimation based on the analyses of application execution trace. The trace is obtained by the execution of the application within an Instruction Set Simulator (ISS). It logs the executed instruction alongside with the memory accesses (with cache missed/hint details). Those activities are then injected in an accurate power model of each architectural components. This approach is not able to take the temporal interactions between components into account. For example,

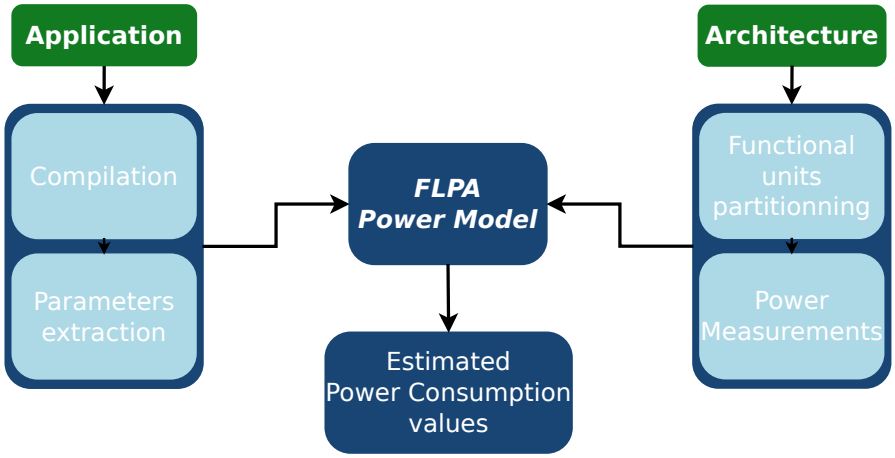


Figure 3-1 – **FLPA principle for estimating power consumption of an application.**

the delay generated by two concurrent accesses on the same memory bank are not taken into account. Those gaps on event modelling could have a major impact on the performance and power accuracy estimation when targeting MpSoC.

1.2-2 Functional-level

The functional-level power model approach uses coarse -grain separation of the target architecture. In fact, the method is to split the target architecture into functional macro blocks, such as instruction type, memory block, etc. Accordingly, the considered events are coarse grain too, so that the estimation time is reduced. Details on the available tools are given in the following.

ILPA: Tiwari *et al.* [Tiw+96] proposed the Instruction Level Power Analysis (ILPA). This approach attempts to correlate power with individual processor instructions. Instead of modelling the utilization in hardware units, power is directly attributed to instructions and how they stress the underlying processor pipeline. The core idea is that the additional effects in terms of inter-instruction dependencies, instruction operand bits, pipeline stalls and cache misses can be modelled on top of the base energy consumption of each instruction. Tiwari's approach is very accurate but requires a lot of analysis and measurements and therefore the development time of a model for a target processor is considerable. The work done by Sinha *et al.* at the Massachusetts Institute of Technology leaded to the Joule Track tool [SC01]. This tool proposes an environment based on Tiwari's approach, which allows power consumption estimation of the StrogARM SA-1100 and Hitachi SH-4 processors from C-written applications.

FLPA: Tiwari's approach has been extended to bypass the complexity of processor model building and to improve its compatibility with complex architectures. For this purpose Senn *et al.* introduced the SoftExplorer tool [Sen+05]. The integrated estimation method is based on Functional-Level Power Analysis (FLPA) [Qu+00; Lau+04]. This coarse grain method focuses on processor macro-functions such as processing units, pipeline stages, internal memories and caches. Thus, it reduces the number of measurements needed to build a processor model and shortens its development time. Fig. 3-1 exposes clearly the two phases involved in this power modelling approach. First, the architecture is split into functional blocks and the power parameters are extracted for each of them (right side of the figure). In parallel, the application execution is monitored to extract the occurrences of events belonging to each functional units (left side of the figure). Then, information is injected

in the FLPA power model and the global power consumption values are computed. As described in [BSN04; LBN05], a good estimation accuracy can be achieved for typical (digital signal) processor architectures.

HLPA: Blume *et al.* extended the FLPA modelling concept to model embedded as well as heterogeneous processor architectures featuring different embedded processor cores. They introduce a so-called hybrid functional-level and instruction-level (HLPA) [Blu+07] model to draw benefits of the two approaches: high accuracy and low modelling effort. This approach was applied to a variety of basic digital signal processing tasks ranging from basic filters to complete audio decoders on the ARM940T architecture. Results shown an estimation error of 9% and allow to efficiently evaluate coding styles, compiler settings, and algorithmic alternatives.

VPPET: VPPET [Ret+14b] is a virtual platform introduced by Rethinagiri *et al.* that enables power and energy estimation on HMpSoC. The estimation is done in two phases. In the first one, a power characterization of the target architecture is done. The second one monitors the execution of the target application and uses the results of the first phase to compute the power estimations. The power characterization of the target architecture is built upon an extended version of FLPA method that targets MpSoC and FPGA fabrics. The power characterization of a platform is a one-time activity. First, the platform is divided into different functional blocks (e.g. Arithmetic and Logic Unit, Load and Stored Unit). Then, parameters are assigned depending on the functionality of the block. There are two types of functional parameters: algorithmic parameters that depend on the executed algorithm (e.g. instruction per cycle, cache miss rate, area utilization) and architectural parameters that depend on the component configuration set by the designer (e.g. clock frequency, bus frequency and number of processor cores). The values of those parameters are obtained by using micro-benchmarking that stimulate each functional block separately for multiple parameter values. The second step monitors the application execution properties with the help of an instruction accurate fast Just In Time (JIT) simulator. Those results are then combined to obtain the power estimation of the target application on the target architecture.

As shown is this Section, a wide range of power modelling tools exists. Different levels of description, with a huge impact on computational time, are involved. Low-level techniques are put aside early, as they model continuous physical phenomenons and are therefore computation intensive. The mid-level approach has drastically decreased the computation time with a low impact on accuracy but is not sufficient for fast DSE. In top of that, the engineering cost for building those power model is high and prevents to target a wide spectrum of architectures. The high-level power modelling tools reduce this engineering cost, as well as the computation complexity. However, most of these models rely on application simulation and are therefore not adapted to fast DSE. In simulation-based approach, each simulation evaluates only one design configuration at a time. So, it does not matter how fast a simulation is, it would still fail to examine many configurations in the design space. The other ones, not based on simulation, mainly target GPP architecture and are not convenient for the modelling of heterogeneous architectures. To close the estimation time gap and address the DSE power modelling requirements, we propose in Chapter 4 an analytical power model focused on communication and task-based computation energy analysis that targets HMpSoC architectures.

2 Computer Aided Design tools

This section introduces CAD tools that enable to map sequential application on manycore architectures. In Sub-Section 2.1, we detail the algorithms that enable to extract the potential parallelism of an application from sequential input codes. Then, Sub-Section 2.2 presents DSE tools that help the user to map the previously obtained parallel application description on a given manycore architecture.

2.1 Parallelism-extraction

Over the years, multiple algorithms and techniques have been developed for extracting parallelism from a sequential code. Most of the early efforts were focused on the extraction of regular loop-nest parallelism. Loop parallelism algorithms rely on some loop transformations that expose more parallelism. Nevertheless we need to keep in mind that parallel loop generation is not sufficient to obtain efficient parallel application (cf. Amdahl's law [Amd67]).

The complexity of target applications and the advent of multi-processor architectures motivated the development of new algorithms capable to produce parallel code that better exploits the computational power available compared to the previous parallel loop generation approach. This section presents recent algorithms and tools that target the transformation of sequential applications into parallel ones.

2.1-1 Algorithms

Integer Linear Programming (ILP): Cordes *et al.* present parallelizing algorithms based on an ILP formulation to extract both task-level and pipeline-level parallelisms. These algorithms extract parallelism from applications represented as an AHTG. With their bottom-up approach, they go through the hierarchy levels one by one, starting with the lowest ones. When the analysis reaches a node at the hierarchical level n, the algorithm already knows the optimal solution for the level $(n-1)$ for parallelism granularity going from 1 to $Task_{MAX}$ which is the maximum parallel execution units available in the target architecture. With the ILP formulation, the optimal solution of level n is computed. For this purpose, the algorithm searches the available parallelism at stage n and combines it with the optimal solution at stage $(n-1)$, varying the concurrent task from 1 to $Task_{MAX}$. To adopt this approach in different hardware platforms, the authors propose that the user can specify two parameters:

- Communication costs are needed if data is transferred from one task to another. The communication overhead can also be changed by the user to model different hardware platforms (communications take place only at the beginning and at the end of a task).
- Task creation overhead is added to the computed path for every created task. This parameter can be used to steer the granularity of the parallelization step, depending on the utilized hardware platform.

These ILP algorithms are detailed in [CMA10; Cor+11; Cor+13b; Cor+13a].

Decoupled Software Pipelining: Ottoni *et al.* presented an automatic parallelism-extraction that uses Strongly Connected Component (SCC) and Decoupled Software Pipelining (DSWP) [Ott+05; Vac+07; Ram+08]. DSWP partitions loop body into stages of a pipeline and exposes a multi-threaded

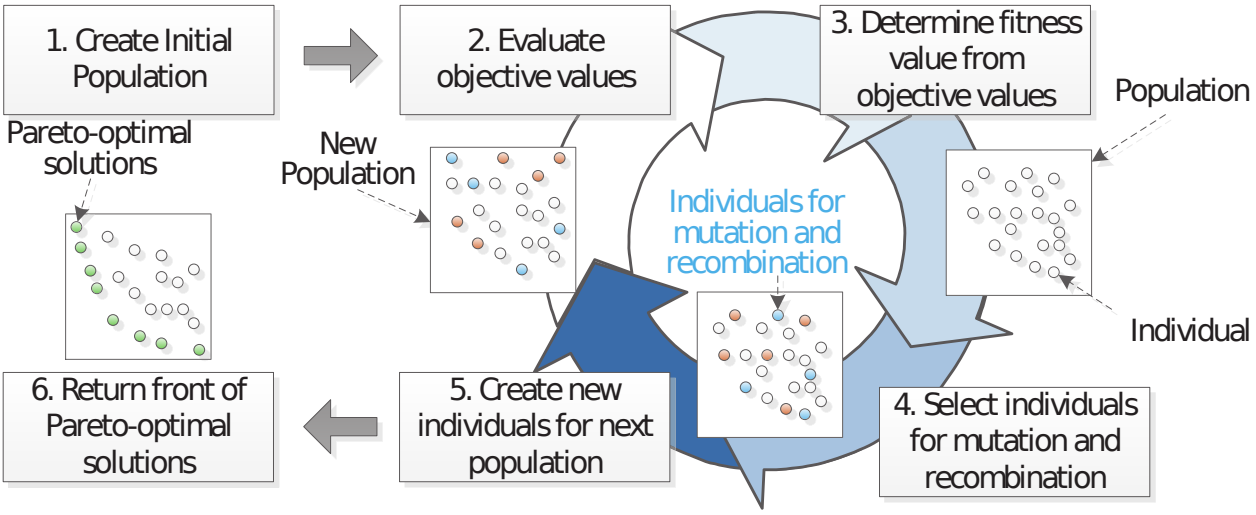


Figure 3-2 – **General structure of genetic algorithms [Cor14].**

pipeline parallelism efficient for many embedded systems. It tries to pipeline programs into dependent communicating threads. DSWP can be efficiently applied on various loops even though they are not using recursive data structures. The obtained pipeline stages are then balanced by on heuristic which merges the nodes with the highest estimated execution cycle. This step is repeated until the estimated cycles of the current step reach the overall estimated cycles divided by the number of pipeline stages. Nevertheless, DSWP operates at assembly level, which drastically limits portability and the readability of the results.

Genetic Algorithms Cordes *et al.* proposed a Genetic Algorithm (GA) for automatic extraction of both the task-level and pipeline-level parallelisms. This algorithm is able to parallelize sequential C applications considering multiple objectives: execution time, communication overhead and energy consumption. GAs facilitate the task of solving optimization problems in a multi-objective manner. Their structure are depicted in Fig. 3-2. The main challenge with the use of GA algorithm is to define gene structures that can efficiently represent multiple objectives. This extraction algorithm is built around three steps. Firstly, the AHTG representation is extracted from the application code. Secondly, the parallelization process starts to parallelize with a bottom-up approach. Each hierarchical node is individually processed and a front of Pareto-optimal is built, with solutions for the different objectives. When all nodes on the same level of the hierarchy are processed, the parallelization algorithm continues with the parent node. The algorithm chooses one of the solutions of the Pareto front of each child node which may contain additional tasks that are deeper in the hierarchy. This procedure continues until each node in the HTG is processed and the top node is reached. Then, the global Pareto front of parallel solutions is returned. The user is in charge of the selection of the best trade-off scenario following his constraints.

Polyhedral model: The polyhedral model is a mathematic representation of iterative program structures. This computational model provides convenient abstractions to apply program transformations that enable to optimize sequences of possibly imperfectly nested loops for parallelism and/or data locality. Although, the complexity of the code generation has been a long-time barrier, Bastoul proposed in [Bas04] a fast and efficient code generation technique that enables the integration of polyhedral framework within compilers for iterative optimization schemes. In [Bon+08], Bondhugula *et al.* go one step beyond and propose an analytical model-driven for automatic trans-

formation within the polyhedral model. This approach is driven by an integer linear optimization framework to find good ways of tiling for extract parallelism using affine transformations. These transformations could be used to generate openMP parallel code from sequential C code.

2.1-2 Frameworks and Tools

Over the years, many tools have been developed to parallelize sequential code. Most of these tools [BGS94] were dedicated to High Performance Computation (HPC) applications and architectures, their applicability to embedded multi-core systems is limited due to the heterogeneity of these systems and their mixed memory structure. In this section, we present some tools targeting the embedded system world. Some of them use algorithms presented in Subsub-Section 2.1-1.

Par4All: Par4All [Ami+12; Tor+12] is an automatic parallelizing and optimizing compiler for C and FORTRAN sequential programs. It is based on the source-to-source compiler infrastructure PIPS [Iri91] and targets architectures such as multi-core systems, HPC systems, GPUs, and some parallel embedded heterogeneous systems. The source-to-source approach of Par4All enables a good interoperability with other tool chains, such as highly optimized vendor compilers for a given processor or platform. The main goal of Par4All is to ease the code generation for parallel architectures from sequential source codes with almost no manual code modification required. Par4All relies on PIPS inter-procedural capabilities like memory effects, reduction detection, parallelism detection, but also polyhedral-based analyses such as convex array regions and preconditions. The current version of Par4All can generate CUDA, OpenCL and OpenMP code from C code with a simple easy-to-use high-level script facility.

PaxES: Parallelism Extraction for Embedded Systems (PaxES) [CM12b] is a parallelization framework optimized for embedded multi-core systems. This tool, developed at TU Dortmund, embeds three extraction methods that target homogeneous and heterogeneous processor-based architectures. It is specifically designed for multi-core embedded systems and considers multiple objectives such as speedup, communications cost and energy. The embedded energy model is not detailed. At a first glance, it seems to associate one computation cost for each task with a communication overhead. The static power consumption is not taken into-account, nor the involved communication channel/processor type. The parallelism algorithms used were presented in Subsub-Section 2.1-1. Fig. 3-3 shows the general structure of the tool. As we can see, PaxEx is included in a global framework and relies on external tools for solving the genetic-algorithm as well as for generating the code.

MAPS: The MPSoC Application Programming Studio (MAPS) [Cen+08; LC10] performs a semi-automatic parallelization technique in which the user can manually steer the granularity of the extracted parallelism. This tool could exploit Task-, Data- and Pipeline-level parallelism. The parallelization process in MAPS is built around three phases: analysis, partitioning and code generation. MAPS takes as inputs the sequential C code of the applications as well as the description of the MpSoC target platform. The analysis phase uses both static and dynamic approaches to extract data and control flow information and generates a Weighted Statement Control Data Flow Graph (WSCDFG) annotated with cost information. Then the MAPS partitioning tool searches for parallel tasks in the target application. The partitioning phase of MAPS is performed with two algorithms: Strong Connected Component (SCC) improvement and load balancing. At the end, the code generation phase translates the resulting tasks into a parallel C code.

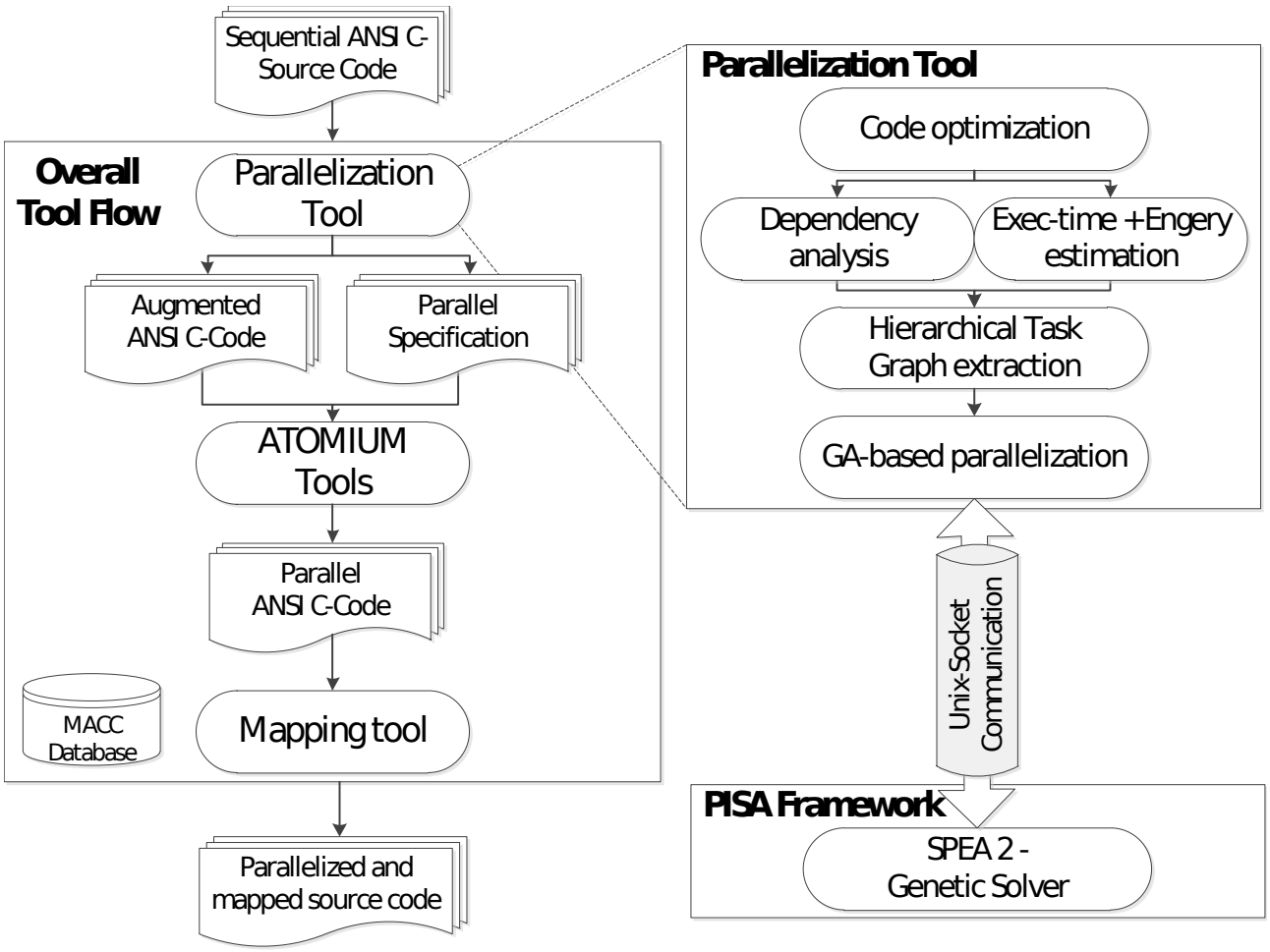
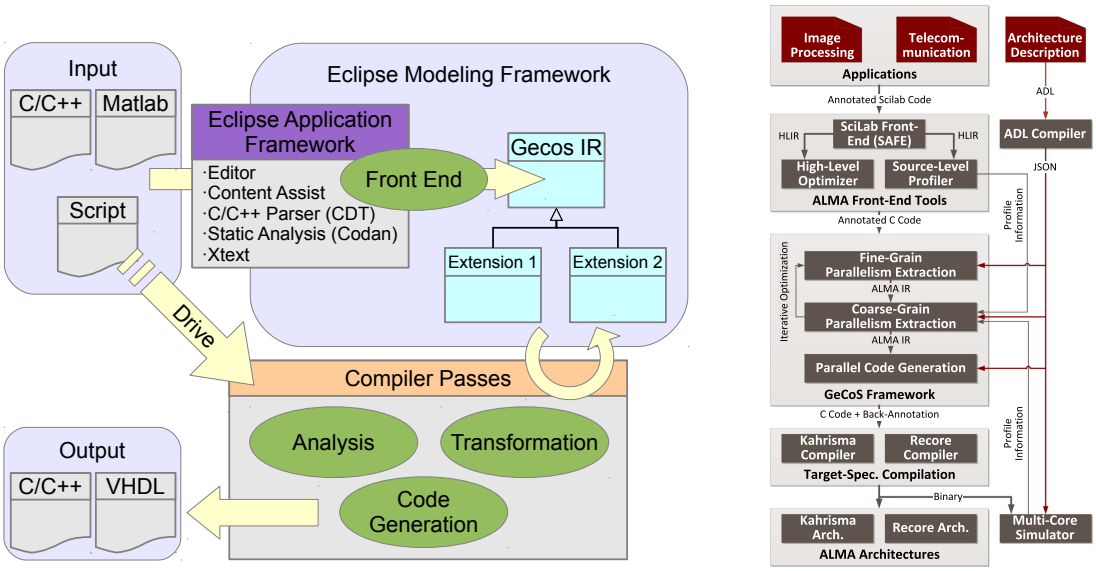


Figure 3-3 – **General structure of PaxES Tool [CM12a].**

GeCos: Generic Compiler Suite (GeCoS) [Inr; Flo+13] is an open source framework developed at IRISA in the CAIRN team which provides a highly productive environment for hardware design. The GeCoS infrastructure facilitates the prototyping of hardware design flows, going beyond compiler analyses and transformations. It enables a good interaction and feedback between the compiler and the designer which is essential in a DSE context. Fig. 3-4a presents a high-level overview of GeCoS and its related tools. The overall flow is similar to a typical compiler; the input is parsed through the front-end, processed via several analyses and transformations, and then outputs are generated from the modified Intermediate Representation (IR).

This highly extendable framework was used within multiple parallelization projects. For example, the ALMA European project [Str+13] used the GeCoS framework to propose a fully integrated tool chain that enables the efficient mapping of applications on multiprocessor platforms from a high level of abstraction, such as Matlab/Scilab specifications [Sci12]. Fig. 3-4b depicts the overall design flow involved in ALMA.

Parallelism extraction is a widely spread topic, started long time ago with the appearance of supercomputeur. With the advent of MpSoC, the memory layout of the target architecture has shifted from distributed- to shared- or even mixed- schemes with SoC embedding a NoC. The parallelism extraction community has widened its spectrum to embedded system architectures and is now able to optimize applications for such memory layout. Some of them target the heterogeneity, but not as defined in this thesis. For them the heterogeneity is defined by the integration of processor cores with multiple ISA and multiple frequencies. The use of dedicated accelerators, such as ASICs and



(a) **GeCoS flow overview [Flo+13]**

(b) **ALMA design flow overview [Str+13]**

Figure 3-4 – **Overview of GeCoS and its use within the ALMA project.**

FPGAs, stays at the frontier of their scope and is not investigated. Furthermore, the power/energy consumption is not well supported in these tools. Cordes *et al.* proposed a multi-objectives approach in [CM12a; Cor+12; CM12b] to tackle this aspect with an analytic power model. However, the model does not fit with the HMpSoC architecture, and no experiments are done to evaluate the model accuracy.

2.2 HMpSoC Design Space exploration

In this subsection, we present multiple DSE tools that target MpSoC, HW accelerators or HMpSoC. These DSE aim at evaluating system level performance and helping the user to find efficient solutions. In the first part, we introduce the simulation-based DSE frameworks that present good exploration accuracy but have an important exploration time. The second part introduces analytical-based frameworks. Finally, the DSE frameworks targeting HW accelerators and HMpSoC design are presented in the last part.

2.2-1 Simulation-based DSE

Sesame: The Sesame [PEP06; Erb07] framework has been developed by Erbas *et al.* within the context of the Artemis project [Pim+01]. It targets system-level performance evaluation and architecture exploration of heterogeneous multimedia embedded systems. For this purpose, Sesame provides high-level modelling, estimation and simulation tools. With the help of Sesame, a designer can explore the design space through high-level system simulations, and gradually lower the abstraction level by incorporating more implementation details to reinforce the accuracy of performance evaluations. Sesame decouples the application from the architecture and use the Y-chart design methodology [Bal97]. According to the Y-chart approach, the functional behavior of an application is described in an architecture-independent manner. The exploration problem in SESAME is multi-objective as it focuses on optimizing performance, power, and cost. Evaluating a single design point

using trace-driven co-simulation, then, exploring the design space of mappings using the genetic solver SPEA2 allows to efficiently prune the design-space.

PARADE: Platform for Accelerator-Rich Architectural Design and Exploration (PARADE) [Con+15] claims to be the first cycle-accurate full-system simulation platform for heterogeneous architectures. PARADE starts by quickly modelling each accelerator with the help of high-level synthesis (HLS) tools. In addition, the PARADE flow automatically generates dedicated or composable accelerator simulation modules. Then, the accelerator resources are managed with hardware global accelerator manager described with cycle-accurate models. These accelerator models are gathered with cycle-accurate model of the coherent cache/scratchpad with shared memory between accelerators and CPU cores, as well as customizable NoC based on the widely used full-system simulator gem5 [Bin+11]. Finally, visualization support are added to the model to help the design space exploration. To summarize, PARADE enhances the existing cycle-accurate gem5 simulator with high-level synthesis and RTL simulation model for the accelerator. Alongside with performance simulation, PARADE also models the power, energy and area using existing toolchains including the power modelling tool McPAT [Li+09]

Open-PEOPLE: Open Power and Energy Optimization Platform and Estimator (Open-PEOPLE) [Ati+13], provide a complete platform to ease the design of complex systems. It aims to allow rapid power/energy estimation for heterogeneous systems and could simulate the effect of multiple optimizations on the global power consumption. The platform embeds power estimation at multiple abstraction levels. It enables simulation refinement with different speed/accuracy trade-offs. These power models are built around monitoring of relevant activities. These activities are characterized using measurements on real boards. Afterwards, power models are elaborated by regression functions or simply recorded as multi-entry look-up tables.

DESSERT: DESign Space ExploRation Tool (DESSERT) [Ret+14a] proposes to software developers a tool to easily port and estimate applications power/energy using all the processor cores of a system such as Exynos5 board. This exploration tool provides power and energy traces using special counters to analyze the consumption of an application and help to find hot spots related to power and energy. DESSERT provides rapid prototyping, accurate power/energy estimation at system-level and helps the fine tuning of applications for a particular processor architecture.

The presented tools propose high-level simulations for multi-objective design space exploration with good accuracy. However, as said before, the major drawback of any simulation-based approach is their inability to cover a large design space. In fact, each simulation evaluates only one design point at a time, so regardless the simulation time, it would still fail to examine many points in the design space.

2.2-2 Analytical-based DSE

PRISMSYS: PRISMSYS [Khe+14] relies on the semantics of the Model of Computation (MoC) alongside with traditional engineering models such as UML (Unified Modelling Language [B+96]) or more specialized one like SysML [FMS08] or MARTE [GS08]. It enables to capture both functional and non-functional properties of a system. On top of that, PRISMSYS provides a joint simulation framework for both discrete and continuous parts of the model. It uses a Clock Constraint Specification Language [And+10], which is a formal declarative language for discrete logical time

specification. Associated with Time Square [DM12], Scilab [CCN06] tools and a dedicated backend, it enables the execution of the discrete part as well as the continuous part of the model.

Ammar *et al.* propose in [Amm+16] a DSE tool aiming at exploring the performance and power consumption of MpSoC systems at design time. The key contribution of this DSE framework is the implementation of an energy-aware scheduling process, that integrates the energy-aware duplication-based EAD [Zon+11] algorithm. This algorithm enables to optimize both the performance and the energy efficiency of MpSoC. This DSE tool extends the MARTE model with power aspects and enables a high-level design entry.

COMPSOC: CompSOC [Goo+13] is a DSE tool dedicated to architecture built with the CoMPSoC platform [Han+09]. It takes Data-Flow applications and enables the mapping generation on the architecture with time guarantees. These guarantees are based on the worst-case execution time and the worst-case memory requirements for each actor which should be provided by the user. The CoMPSoC architectures are built with a tiled layout gathering processing elements and memory with a NoC. Each processing tile has a dedicated MicroBlaze [Xil06] processor running the CompOSe real-time management OS. This OS provides various services such as scheduling and power management.

These analytical-based DSE techniques tackle the previous simulation issues and enable to widely explore the application design space. However, some of these models need a description of the targeted architecture as well as applications within a custom modelling structure that could not be generated by the state-of-the-art parallelism extraction tools. The others target a small scope of many-core architectures, which do not really correspond with the current architectural trend introducing heterogeneity.

2.2-3 Hardware focused DSE

ALADDIN: Shao *et al.* introduce in [Sha+15] a pre-RTL accelerator modelling framework for power and performance exploration. This framework, called Aladdin, estimates performance, power and area of accelerators within 0.9, 4.9 and 6.6 percents, respectively, compared to equivalent RTL implementations. It takes accelerator design parameters alongside with a high-level description of the algorithm and then computes the achievable power, performance, and area of the accelerator. Aladdin can be used as on early-stage accelerator design explorer to quickly navigate the large design space of accelerators before the RTL implementation. This approach could greatly reduce the design iteration and thereby the time-to-market of accelerator-rich SoC. Coupled with general-purpose core and memory within an architecture-level simulator, Aladdin can provide a fast and accurate way to model accelerators' power and performance and ease the evaluation of the interaction between accelerators and shared resources in a SoC.

Lumos: Lumos [WS13] is a pre-RTL design-space exploration framework that targets accelerators-rich heterogeneous architectures. It enables to determine the best composition of conventional cores, ASICs accelerators, and reconfigurable logic. Lumos uses a first-order analytic model that extends previous models from [Chu+10]. This framework integrates a fine-grained voltage-frequency scaling model built with Spice simulation and a statistical workload model. Lumos+ [WS16] introduces a

genetic search algorithm to explore the accelerator allocation that maximizes the throughput objectives for a set of workloads that overpass the previous expensive brute-force search. This extension work also enhances the modelling of memory hierarchy and reconfigurable logic.

DSE is an old topic addressed by a large set of studies. In the past, some works have introduced accurate tools based on a simulation approaches. However with the advent of the many-core era, and the explosion of the design space size, the simulation-based approach is no longer an efficient solution. Some analytical methods were already proposed but they rely on dedicated description languages or architectures that narrow their scope and prevent their used with legacy applications. Some tools dedicated to hardware component exploration propose to quickly show the pre-RTL achievable performances. In the following of this document, we will introduce new approaches to reduce the previous exposed gap and make the first steps towards energy-aware design space exploration in HMpSoc architectures.

Toward an Energy-Aware Design Flow [Contributions]

The second part of this work details the main contributions of the thesis. We start by presenting the long-term objectives of this work. For this purpose, we present a high level view of an energy-aware design flow that will enable designers to address new heterogeneous many-core architectures. The main idea is to integrate a power model early in the design flow in order to rapidly explore the design space. The proposed approach independently considers the impact of involved communications between processing cores and the impact of computations in order to propose an analytical formulation that decreases the energy consumption estimation time.

Global design flow

Fig. 3-5 depicts the global design flow targeted as a long term objective in this thesis. As we can see, it could be subdivided in four parts. The first one (A) targets the parallelism extraction, it takes a sequential description of an application and extracts a parallel task graph. The second part of the flow (B) highlights the communications occurring within the task graph, creates a hierarchic level around BpB (cf. Section 4) and generates an EPDG. The third step of the design flow (C) uses architecture properties to enrich the EPDG with execution time and energy of each BpB. Finally, the design space could be explored in the last step (D) with an energy-aware approach to obtain a Pareto Front of the energy versus throughput.

Parallelism extraction (A): This first step takes sequential software as input and uses the state-of-the-art parallelism extraction tools to extract a parallel task graph (cf. Sub-Section 2.1). This is an iterative step that enables the users to tweak the sequential input source to enhance the parallelism extraction results. This step outputs a parallel task graph which roughly exposes all detected parallelisms. When the user is satisfied, the next step is launched.

EPDG creation (B): This second step takes a parallel task graph as input alongside with the parallelism degrees of the target architecture. With this information, this step computes the communications size of each task and merges small tasks or adjacent communication intensive ones in macro tasks. These macro tasks are then extended with two extra nodes (*CommIn* and *CommOut*) to obtain the BpBs. Then, the BpBs are enriched with the communication size and linked together to obtain an EPDG. At this step, the EPDG has no information about the computation energy cost and execution time.

Energy estimation (C): At the beginning of this step, we have an EPDG with no cost information and no idea of BpB potential implementation performance. This step relies on external tools. First, it starts using tools such as Aladdin [Sha+15] to get a fair idea of the potential performance and cost that BpB HW implementations could meet. The user could also use HLS tools, or manually designed solution, to obtain performance values of BpB HW implementations. Then, the obtained performance values are inserted in the design flow to enrich the EPDG.

When all BpB are augmented with performance and cost information, a power estimation tool is called. This tool estimates for each BpB the energy consumption and execution time. These values are needed for each HW implementation and for each available SW core type. Then, all these performance estimation values are inserted in the EPDG and the DSE step can begin.

Design space exploration (D): This step receives as input an EPDG containing communication information alongside with energy consumption estimation for each available BpB implementation.

This step iterates in the design space following heuristic or other optimization algorithms to obtain valid mapping solutions. It uses a fast analytical power model formulation to compute the overall energy consumption of the obtained mapping. At the end, this step outputs Pareto Front of the mapping solutions ordered by overall energy consumption and throughput. Then, the user needs to select the solution in this Pareto Front that fits with his constraints.

In the proposed design flow, the three first steps are based in majority on state-of-the art solutions. In the following, we focus on DSE and energy estimation and propose tools to enable fast and energy aware DSE. For this purpose, Chapter 4 introduces a communication-based power model able to aggregated BpB energy consumption with task mapping information to quickly get the overall application energy consumption. This chapter details a method for extracting the communication energy parameters needed by the power model. The extraction method and the power modelling approach are then validated on a real architecture. In Chapter 5, the previous power modelling approach is used alongside with an optimization method to propose an energy aware accelerator exploration for tiled applications. This method is validated on a real architecture with two computation kernels. These two chapters highlight the difficulties related to the DSE algorithm design. In fact, the validation of these algorithms on a broad range of architectures and applications is hard and time consuming. To address these issues, Chapter 6 proposes an emulation platform integrating a power estimation tool and able to address a large set of HMpSoC architectures. This emulation platform is coupled with an execution framework able to generate an infinite set of representative applications executable on the emulation platform.

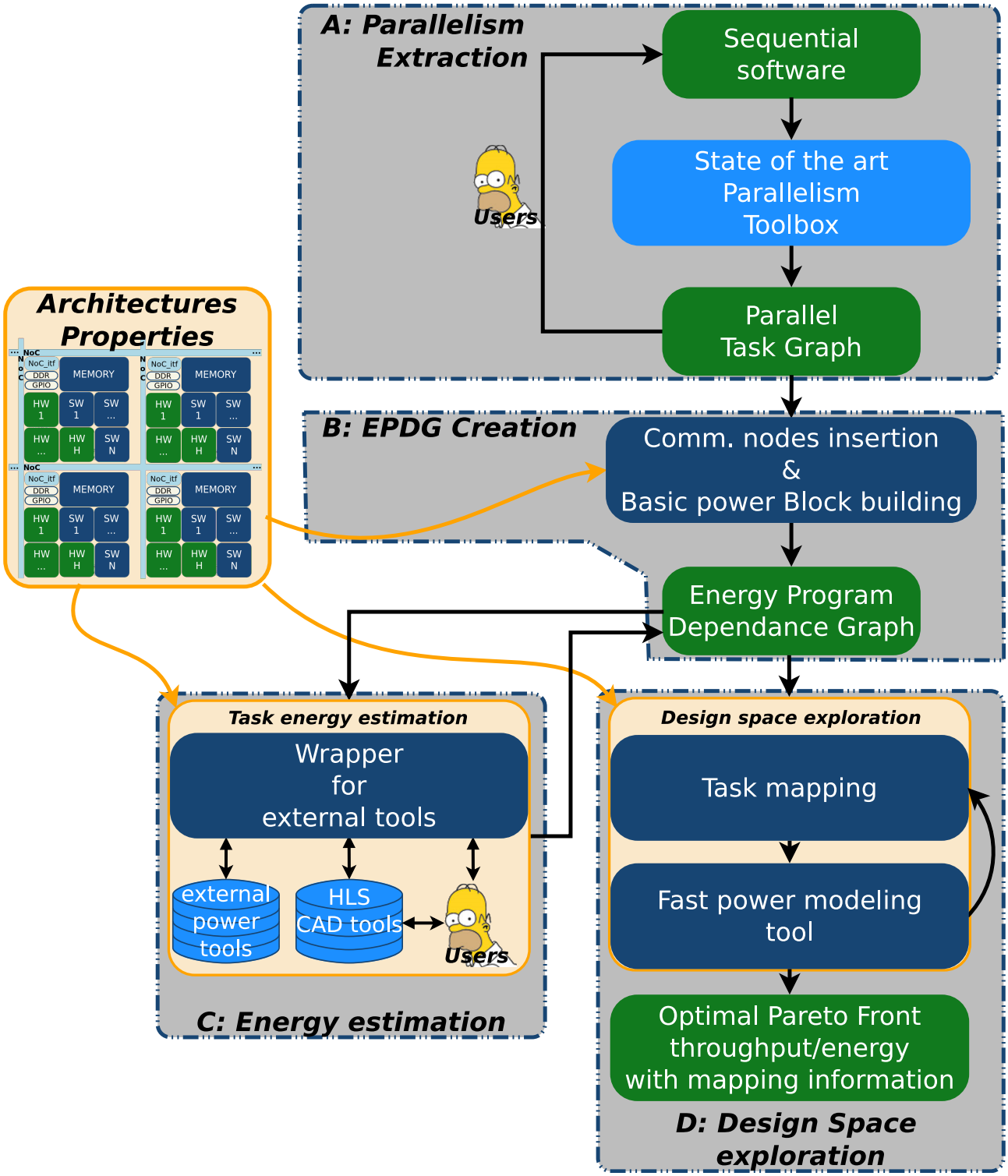


Figure 3-5 – **Global overview of the targeted Energy-aware Design Flow.**

Chapter **FOUR**

FAST POWER MODELING FOR HMPSOC

Contents

As shown in Chapter 3, some high-level power modeling tools have a decent computation time but they rely on application simulation that prevent their use in a DSE environment. Furthermore, these approaches are not initially designed to target many-core and heterogeneous architectures. In this section, we discuss a power modelling approach that circumvents these drawbacks. For this purpose, we extend the model with communication costs and target execution core information, which opens up the model to target many-core architectures. We also decouple the computation cost from the application mapping. With this approach, simulation is needed only once for each elemental task of the target application. Then, the design space exploration and task mapping could be computed with an analytical formulation that fasten the power exploration. This chapter is organized as follows. Section 1 introduces the core of our fast power model. Section 2 shows how to determine the parameters of our power model using micro-benchmarks. This section also presents in depth two architectures (MPPA and Zynq) and displays the micro-benchmark results when applied on real architectures. Section 3 presents the validation of the model on a real HMpSoC architecture using the Zynq. Finally, conclusion are given in Section 4.

1 Communication-Based Power Model

This section introduces the structure of a fast power model for task mapping exploration. This model relies on the memory-centric view of heterogeneous architectures introduced in Chapter 2 Sub-Section 2.2. Considering an application which can be perfectly parallelized, its execution on multiple threads distributed on multiple processors reduces the execution time but not the total amount

of computation. Computations are equally distributed between the processors and therefore the amount of computation can be considered as independent of the parallelism degree. On the other hand, when multiple threads are computed in parallel, the amount of communications and synchronizations is directly linked to the number of execution threads. To formalize this observation, let $A(n)$ be an application iterating n times on k operations. Computational cost $Op()$ of $A(n)$ executed in a sequential manner is:

$$Op(A(n)) = n \times k. \tag{4-1}$$

The parallel execution cost $A_p(n, p)$ of $A(n)$ on p identical threads is:

$$\begin{gathered} Op(A_p(n,p)) = p \times T(n), \\ \text{with } T(n) = k \times \tfrac{n}{p}. \end{gathered} \tag{4-2}$$

We can deduce that the complexity of a perfectly parallelized application is independent of the number of execution threads:

$$Op(A(n)) = Op(A_p(n,p)). \tag{4-3}$$

Assuming that each thread is executed on the same type of processor, the energy consumed for executing the application in its sequential and parallel versions solely differs from the communication cost (and slightly from the static power). Therefore, a power model which is able to evaluate the communication cost and execution time to quickly derive the power consumption of a parallel application is essential for a fast design exploration of the mapping space. In the following, we propose and validate a communication-based power model, suitable for HMpSoC, with a fine-grain resolution and a low computational cost.

The energy consumption of an application executed on a heterogeneous multicore architecture depends on three main sources: the dynamic energy consumption used for computations, the static energy dissipated during execution time, and the energy used for communications between processing cores.

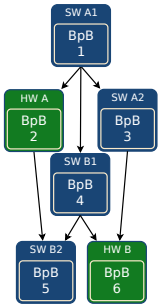
Any parallel applications can be divided into N_{BpB} concurrent computational blocks that could be executed in parallel. Those BpBs (cf. Section 4), represented as nodes, are linked together into a graph, where edges represent communications between blocks. Execution of blocks is atomic: synchronization mechanisms and I/O accesses are held on communication edges. For each pair of BpB, the amount of required communications is considered as known. As the content of these blocks is sequentially executed, each BpB BpB_k contains the same amount of computations regardless of its mapping in the graph. Thereby the energy used for computation in each block could be calculated as well as its execution time. After calculation of these values, the task mapping space can be explored over the N_{BpB} blocks. For each graph corresponding to a mapping solution, the total energy E_t is

$$E_t = E_{stat} + \sum_{k \in N_{BpB}} E_{comp}(BpB_k) + \sum_{(i,j) \in N_{BpB}^2} E_{com}(BpB_i, BpB_j), \tag{4-4}$$

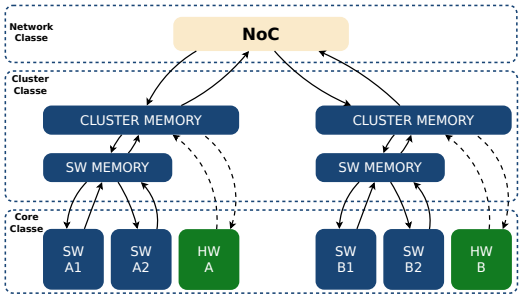
where E_{stat} is the static energy consumption, $E_{comp}(BpB_k)$ is the energy of computations in block BpB_k and $E_{com}(BpB_i, BpB_j)$ is the energy used for communication between blocks BpB_i and BpB_j.

1.1 Computation energy cost

The energy consumption $E_{comp}(BpB_k)$ of each BpB is supposed to be known. In the case of heterogeneous architectures, $E_{comp}(BpB_k)$ is computed for each kind of available computational cores. Since this step is executed only once, the estimation time required by the power evaluation



(a) **BpB graph.**



(b) **Target HMpSoC memory hierarchy.**

Producer	Consumer	Memory Bank Used
BpB_1	BpB_2	Cluster Memory A
BpB_1	BpB_3	SW Memory A
BpB_1	BpB_4	NoC
BpB_2	BpB_5	NoC
BpB_3	BpB_6	NoC
BpB_4	BpB_5	SW Memory B
BpB_4	BpB_6	Cluster Memory B

(c) **Upper communication channels needed.**

Figure 4-1 – **Example of communication channel extraction for a 6-block application.**

tools is not an issue. In the experiments of this thesis, theses values will be directly measured on the real SW/HW execution. Other energy estimation tools for the computation part can be used or adapted from the state-of-the-art [Tiw+96; Qu+00; Lau+04; Blu+07; Ret+14b].

1.2 Communication energy cost

For a given task mapping, the communications between blocks depend on their allocation to a specific memory hierarchy level. For example, the BpB graph represented on Fig. 4-1a, composed of six blocks with two of them being HW compatible, is considered. This BpB graph is mapped over the target HMpSoC shown on Fig. 4-1b, which is composed of two clusters connected through a one-channel NoC. Each cluster contains two SW cores and one HW core, associated with a two-level memory hierarchy. Following the task mapping, the memory bank used for each communication flow can be inferred. Fig. 4-1c shows the communications needed for our example.

When communications are mapped into memory, a solving function can be used to compute the communication cost. Let $C(BpB_i, BpB_j)$ be the set of communication channels crossed from block BpB_i to block BpB_j via the required memory bank. The communication energy can be expressed as

$$E_{com}(BpB_i, BpB_j) = \sum_{c \in C(BpB_i, BpB_j)} e_{0_c} + e_{1_c} \times bytes(BpB_i, BpB_j), \tag{4-5}$$

where e_{0_c} and e_{1_c} are energy parameters of the c^{th} crossed channel and $bytes(BpB_i, BpB_j)$ returns the number of bytes communicated from BpB_i to BpB_j.

In the same way, the communication time T_{com} can be computed as

$$T_{com}(BpB_i, BpB_j) = \sum_{c \in C(BpB_i, BpB_j)} t_{0_c} + t_{1_c} \times bytes(BpB_i, BpB_j), \tag{4-6}$$

where t_{0_c} and t_{1_c} are the crossed channel time parameters. Sub-Section 2.1 introduces a method to determine those parameters for a target architecture.

1.3 Static energy cost

Once the execution time of each BpB and each communication is determined, the overall execution time T_{exec} can be computed as the critical path in the mapping graph weighted with execution

time of computation and communication. Then, the static energy consumption can be deduced as

$$E_{stat} = T_{exec} \times P_{stat}, \tag{4-7}$$

where P_{stat} is the static power consumption that could be measured with the micro-benchmark based method presented in the next section.

2 Deep-dive on two Architectures

The proposed communication-based power model relies on HW parameters of the target architecture that are not necessarily provided by chip manufacturers. These parameters are e_{0_c} and e_{1_c} used in (4-5), t_{0_c} and t_{1_c} used in (4-6), and P_{stat} used in (4-7). In the following, a method based on micro-benchmarks, that enables the extraction of the previous communication-based power model parameters on real hardware multicore architectures, is presented.

We then detail the structure of two architectures briefly presented in Sub-Section 1.2 (MPPA and Zynq). A special attention is paid on the memory hierarchies and the communication costs. Then, the micro-benchmark method is applied on these architectures. The power characterization relies on a set of experiments done on real evaluation board. The experiments aim to extract a cost function for the communication that occur on different communication channels.

2.1 Micro-benchmark principle

A micro-benchmark is a simple and synthetic application that aims at stressing a specific part of the execution architecture. We want to extract an analytic cost function for the communication. The energy and time used by a communication could be represented as a multi-parameter function in which each parameter represents a crossed channel. To obtain the value of those parameters, a solution is to compute the partial derivative for each parameter. This is the aim of micro-benchmarks. To fulfill this purpose, each micro-benchmark is designed to focus on a specific communication channel or a specific memory bank following the following properties:

- Selectivity: micro-benchmarks only stress a specific communication channel, as much as possible.
- Intensity variability: micro-benchmarks stress a communication channel with different intensity (*i.e.* communication size in our case).
- Duration: micro-benchmarks execution time adjust to power-measurement tool resolution (*i.e.* "scaleFactor" in our case).

2.1-1 General structure of a micro-benchmark set

Using the generic architecture depicted in Fig. 4-1b, each cluster is composed of four kinds of communication channels: SW Core to SW Memory; SW Memory to Cluster Memory; HW core to Cluster Memory and Cluster to Cluster through NoC. To determine the model parameters of this architecture, a set of micro-benchmarks composed of four subsets must be built as follows:

- SwChannel: this subset focuses on communication cost between processing core and SW memory level. It generates read or write accesses in an array allocated in SW memory.

```
Data: scaleFactor, size
initBenchmarkEnv()
startPowerMeasure()
for iteration in scaleFactor do
    openCommunicationChannel()
    producer = spawnProducerThread(size)
    consummer = spawnConsumerThread(size)
    waitThread(producer, consumer)
    closeCommunicationChannel()
end
stopPowerMeasure()
writePowerMeasureToFile()
```

Algorithm 1: Generic micro-benchmark structure.

- HwChannel: this subset focuses on communication cost between HW accelerators and cluster memory. It generates read or write accesses in an array allocated in cluster memory.

- IntraCluster: this subset focuses on communication cost between SW memory and cluster memory. Theses parameters can not be directly measured. The micro-benchmark generates read or write accesses in an array allocated in cluster memory from the processing core and then deduces IntraCluster parameters by subtracting the SwChannel value.

- InterCluster: this subset focuses on communications between clusters and generates data transfers through the NoC.

All these micro-benchmark executions are parameterized with the size of the data to communicate. Communication time is an order of magnitude smaller than the usual power measurement time resolution. To overcome this issue and limit measurement noise, it is necessary to build the micro-benchmarks over large number of communications. For this reason, micro-benchmarks are composed of three parts: opening, kernel, and closing. The opening part is responsible of SW and/or HW initialization, and then the micro-benchmark iterates *scaleFactor* time on the kernel. The kernel part generates a communication of *size* bytes over the target channel. Then, the closing part retrieves power measures and logs them into a file. Algorithm 1 presents a generic micro-benchmark structure.

2.2 Kalray MPPA

The MPPA was introduced by Kalray in November 2012. Kalray is a French company created in 2008 focusing on the development of a new high-performance technology for the embedded market. With the announcement, in May 2017, of the third generation of MPPA Coolidge, we see that the company slightly shifts to heterogeneous architecture. In fact, instead of the MPPA-1024 announced 2 years ago, the Coolidge generation will feature 80 up to 160 VLIW processors coupled with 80 up to 160 ASIP dedicated to computer vision and deep-learning claiming peak performance of 5 TFLOPS under 20 W of power consumption. In the following, we focus on the first generation of the Kalray MPPA called Andey.

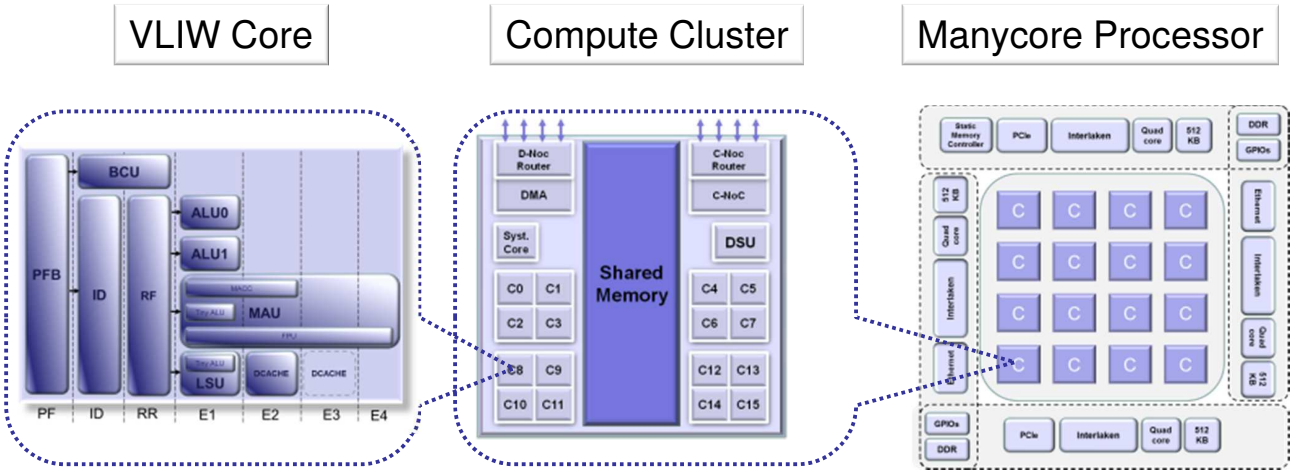


Figure 4-2 – **Overview of the Kalray MPPA parallelism level [Din13].**

2.2-1 MPPA structure

The MPPA-256 from Kalray features 256 VLIW processors gathered in 16 clusters. Each cluster has 2 Mbytes of shared memory and an extra VLIW processor to manage cluster's resources. This architecture is constructed to exploit multiple parallelism granularity. Fig. 4-2 shows an overview of the MPPA hierarchies. As we can see, within the VILW architecture, the instruction-level parallelism will be exploited. Within a cluster, the thread-level parallelism will be used. Finally, multiple processes or applications could run in parallel, one on each cluster.

VLIW architecture: The used VILW architecture features two Arithmetic-Logic Unit (ALU), a Floating Point Unit (FPU), a Multiply-Accumulate Unit (MAU), a Load-Store Unit (LSU), and a Branch-Control Unit (BCU), which makes it a 5-issue architecture. Furthermore, this architecture relies on a 7-stage instruction pipeline running at 400 MHz. To enhance the execution time predictability, the cores use Least Recent Used (LRU) cache with a low miss penalty.

Memory architecture: The MPPA has 20 distinct memory spaces, 16 in the clusters, 4 in the I/O subsystems that enable access to the external DDR memory. The used NoC has two distinct paths: a high throughput one dedicated to data transfer, and a low-latency one for control purpose. Clusters are linked together through a $2D$-Torus topology (Cf. Fig. 2-4b). So, the oriented-tree memory hierarchy of the MPPA is composed of a two-level network class (one for throughput, one for latency). The cluster class is separated into two categories: the first one is the standard cluster and consists of one-level cluster class, the second one is the IO cluster and consists of a three-level cluster class. The core class, which represents the cache hierarchy, is a one-level class (cache L1). The global structure of the oriented-tree memory hierarchy is depicted on Fig. 4-3.

Experiment infrastructure: Experiments are based on the MPPA developer (AB01) evaluation kit. It features an x86 architecture hosting the Kalray toolchain and a PCIe board with the MPPA. This infrastructure has a power measurement facility, called *k1-power* that measures the overall power consumption of the architecture at a sample rate of 50Hz. On the first board generation, the *k1-power* tool measures only the global power supply before a dynamic voltage adapter (International Rectifier IR3550) that induces an extra power loss depending of the power dissipation. Fig. 4-4 shows the power loss and efficiency of this component. *k1-power* uses this power efficiency curve to correct the power consumption output. As we can see, the power facility available on the MPPA

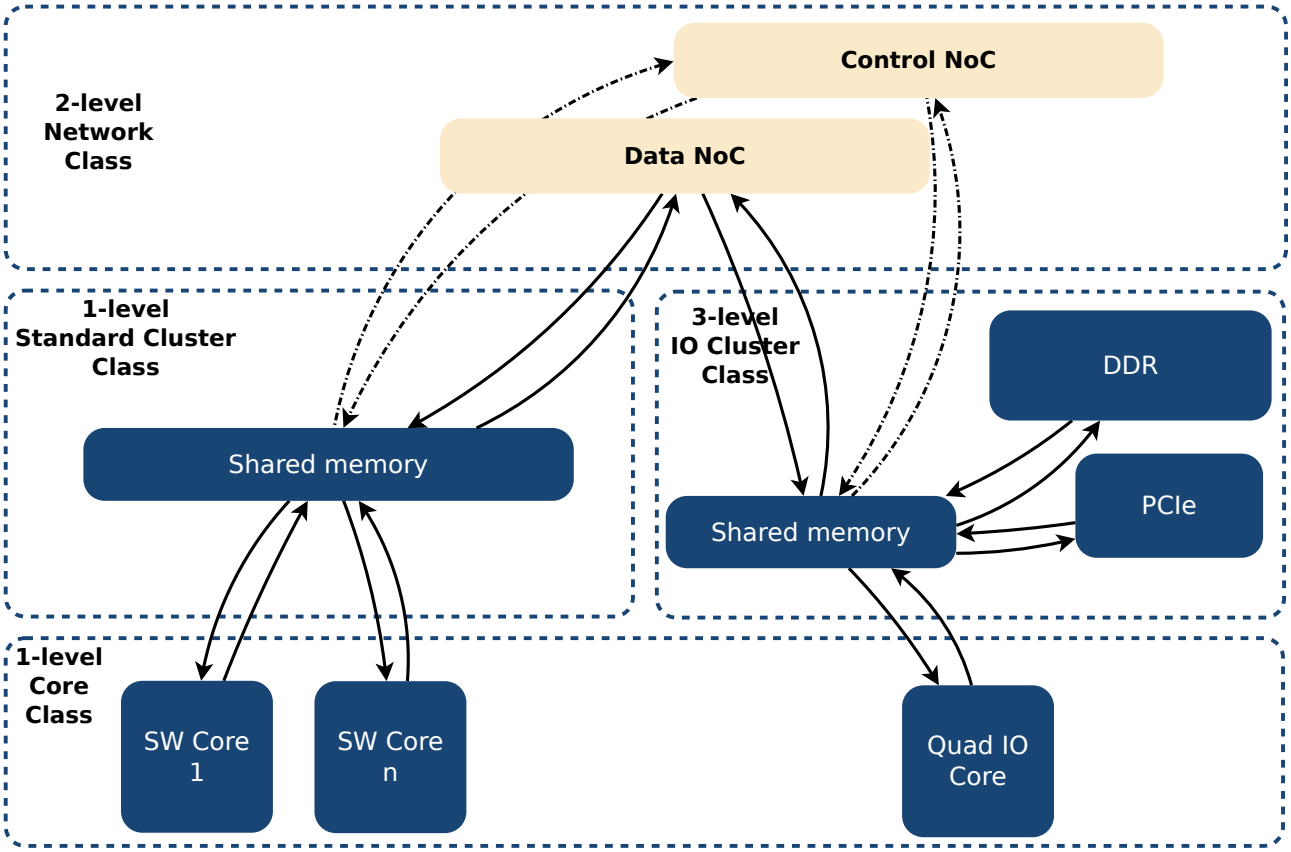


Figure 4-3 – **Details of the Kalray MPPA memory hierarchies.**

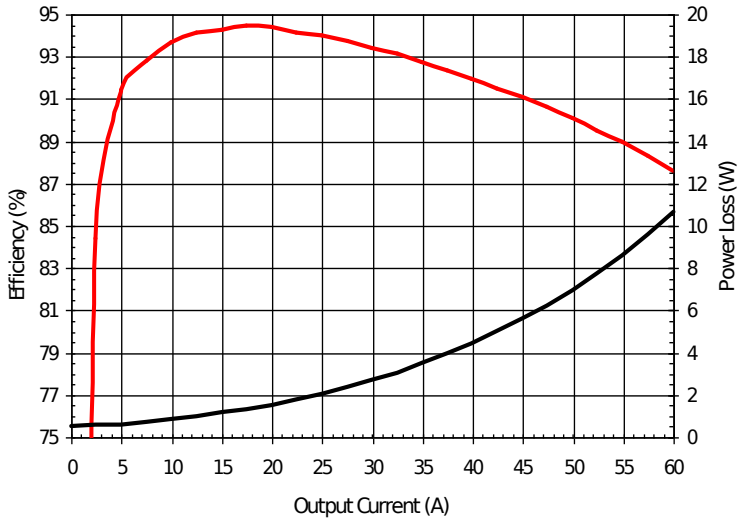
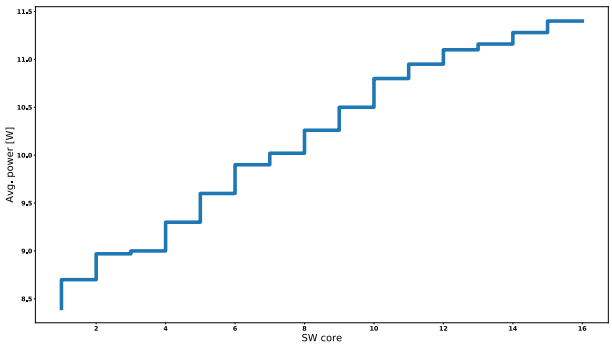


Figure 4-4 – **Typical IR3550 Efficiency and Power Loss.**

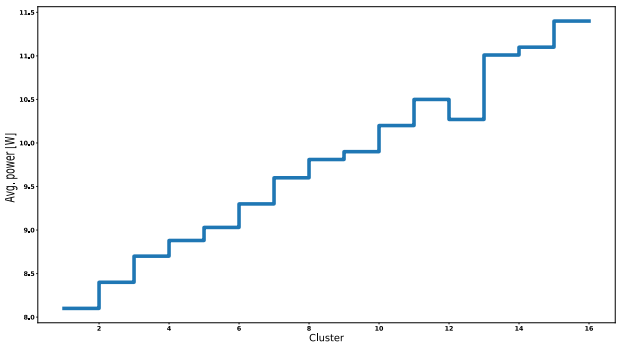
seems not to be very accurate but we were able to mitigate the error and the low sample frequency by extending the micro-benchmark execution time.

2.2-2 Power analysis

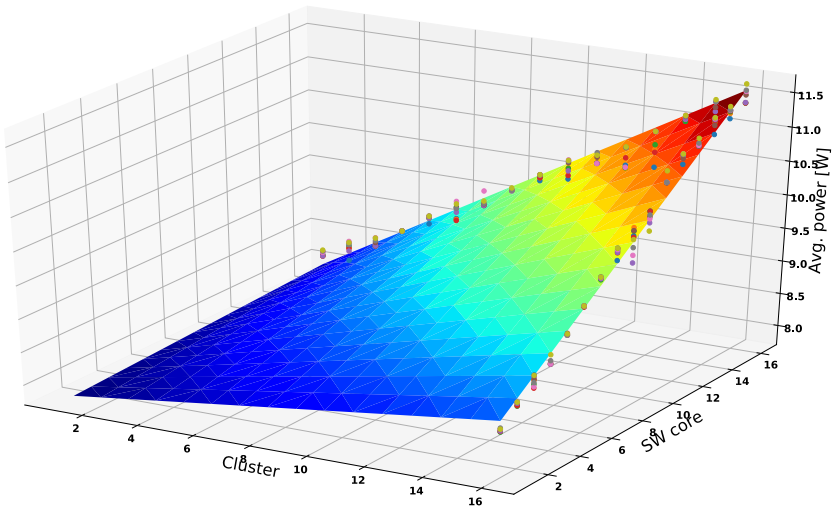
In this subsection, we investigate the power consumption of the Kalray MPPA. The main purpose is to isolate the power incidence of communication over the available communication channels. We



(a) **Avg. power = f($N_{SW_{core}}$)** $N_{cluster}$**=16**



(b) **Avg. power = f($N_{cluster}$)** $N_{SW_{core}}$**=16**



(c) **Measured and estimated power = f($N_{SW_{core}}$, $N_{cluster}$)**

Figure 4-5 – **Static power consumption.**

start by writing a small set of 5 benchmarks, as detailed below:

- Static: this micro-benchmark measures the MPPA static consumption depending on the number of cluster/SW cores loaded with binary.
- ComSWMem: this micro-benchmark targets channels between the SW cores and the cluster memory level. It generates read or write operations from SW core in an array located in the cluster memory.
- SyncSW: this micro-benchmark evaluates the synchronization mechanisms between the SW cores within a cluster.
- Cluster: this micro-benchmark targets data communication within the NoC. It generates transaction between pairs of clusters.

Each micro-benchmark listed above is run on multiple configurations. These configurations are obtained by varying the communication size, the number of SW cores and/or clusters involved.

The first micro-benchmark focuses on the static power consumption implied by the number of clusters ($N_{cluster}$) and the number of SW cores (SW_{core}) started. For this purpose, a set of applications generating wait statements is launched over the MPPA. For each test, the number of SW cores enabled within clusters and the number of clusters are varying. Fig. 4-5a shows the power consump-

MPPA static power coefficient	
P_{Base}	7.7281 W
$P_{Cluster}$	0.0391 W
P_{SW}	0.0126 W

Table 4-1 – **Extracted static power coefficient of Kalray MPPA.**

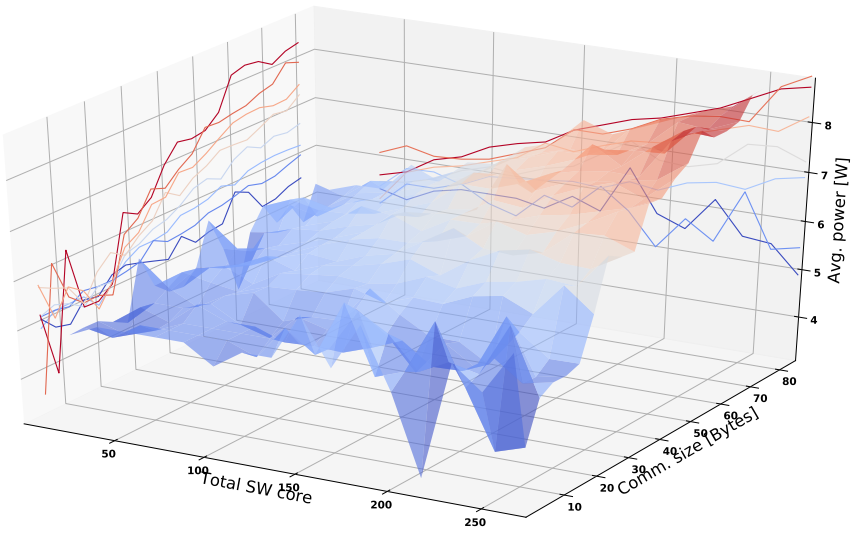
tion variation following the number of SW cores used within the 16 clusters. Figure 4-5b shows the power consumption variation over the number of clusters used with the 16 SW cores activated in each of them. In spite of some strange measurements results, such as when 13 clusters are enabled on Fig. 4-5b, we can extract the static power coefficients of the MPPA as follows:

$$Power(N_{SW_{core}}, N_{cluter}) = P_{Base} + N_{cluster} \times (P_{Cluster} + (P_{SW} \times N_{SW_{core}})). \tag{4-8}$$

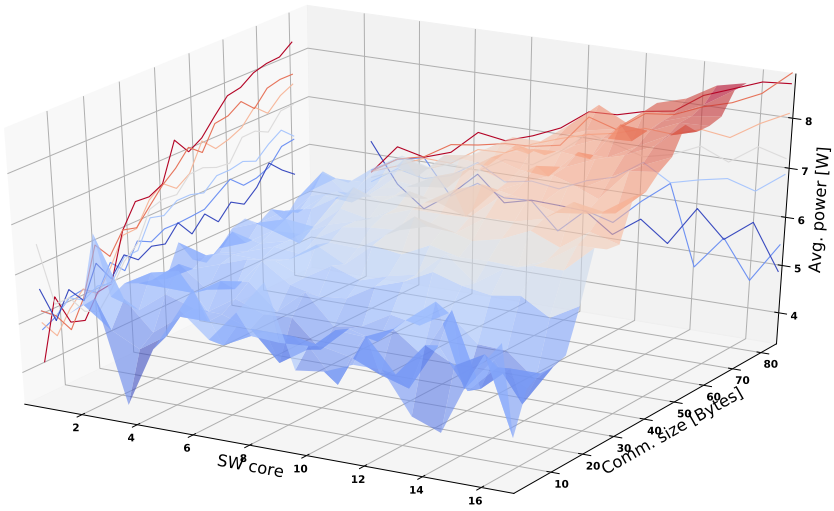
The obtained parameter values are displayed in Table 4-1. These values are used in Fig. 4-5c to draw the estimated statical power on all available configurations. The MPPA power consumption goes from 7.7 W when none of the standard clusters is launched, up to 11.6 W when all SW cores are launched in all clusters.

Fig. 4-6 and Fig. 4-7 display the average power consumption and time of communication between SW core and cluster memory. Fig. 4-6 focuses on read access. The 3D plot displays the average power consumption following the number of SW cores with a variation scale from one SW core for Fig. 4-6a to 16 SW cores for Fig. 4-6b. The obtained behavior does not match with the expected one. In fact, it seems weird that the communication size has an impact on average power consumption. We expect an incidence on the execution time and so on the energy consumption but not on the average power. In addition, the observed curves present a lot of spikes and noise. Then, we decided to further investigate the *k1-power* outputs. We observed a really low quality of service of this module. In fact, the outputs could randomly be stuck at 0 (observed up to 6 times in a row in our experiments). This could drastically degrade the sample frequencies and reduce our confidence on the tool accuracy. To investigate further, we eliminated the false measured point. Fig. 4-6d and Fig. 4-6c show a projection of the previous curves on a 4 Byte communication size. The erroneous points were removed from the experiments, but these curves always present weird results. These elements make us think that the *k1-power* is really buggy on our experiment infrastructure, and could not fulfill with our accuracy requirements. We investigated deeper with another benchmark set that targets write access between SW core and cluster memory. Results are depicted in Fig. 4-7 and show the same inaccurate results.

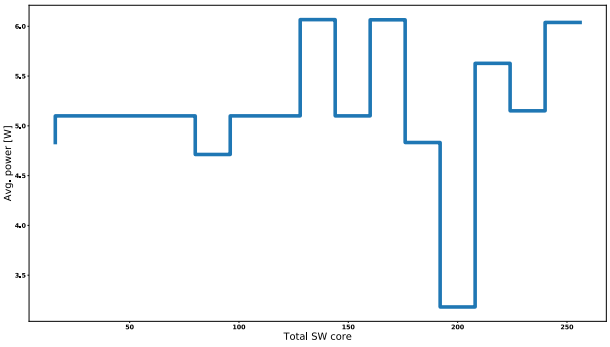
After these experiments, we decided to put aside the Kalray MPPA due to is poor power measurement accuracy and we focused on another architecture: the Xilinx Zynq.



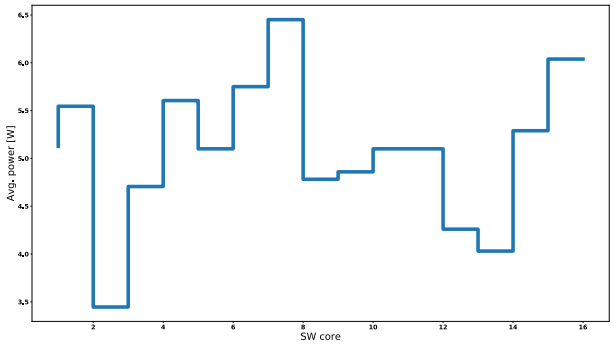
(a) **Avg. power = f(Size, Total $N_{SW_{core}}$)**



(b) **Avg. power = f(Size, $N_{SW_{core}}$) C=16**

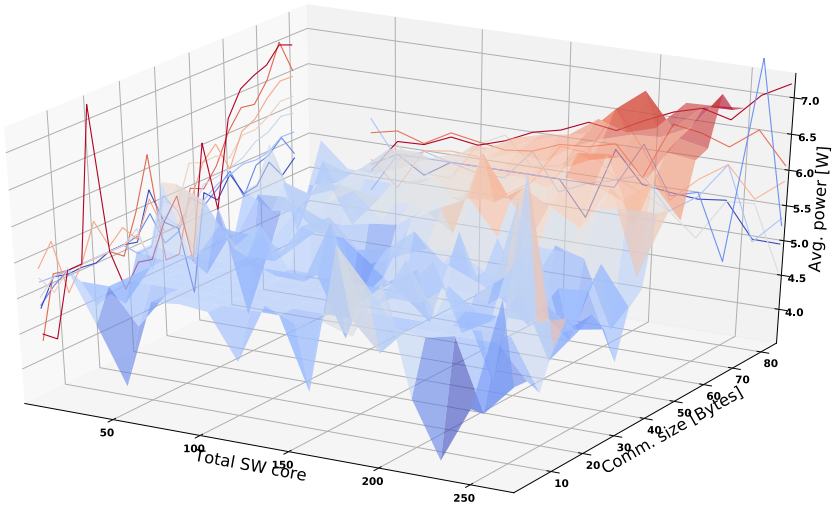


(c) **Avg. power = f(Total $N_{SW_{core}}$)**

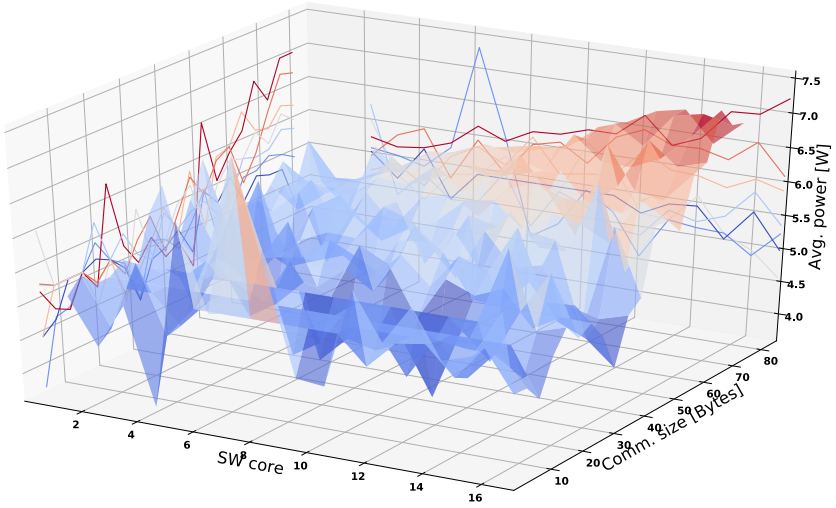


(d) **Avg. power = f($N_{SW_{core}}$) C=16**

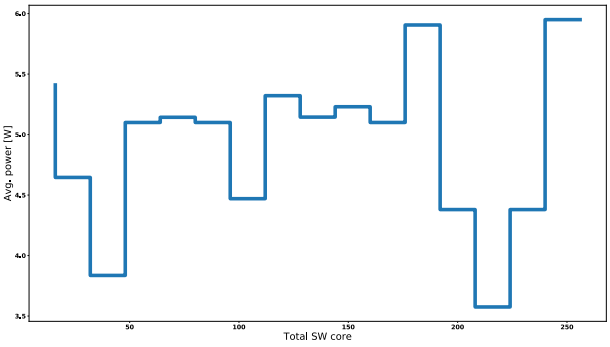
Figure 4-6 – **Power consumption of read access within cluster memory.**



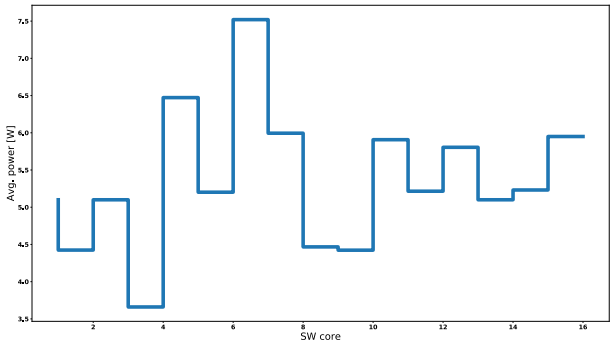
(a) **Avg. power = f(Size, Total $N_{SW_{core}}$)**



(b) **Avg. power = f(Size, $N_{SW_{core}}$) C=16**



(c) **Avg. power = f(Total $N_{SW_{core}}$)**



(d) **Avg. power = f($N_{SW_{core}}$) C=16**

Figure 4-7 – **Power consumption of write access within cluster memory.**

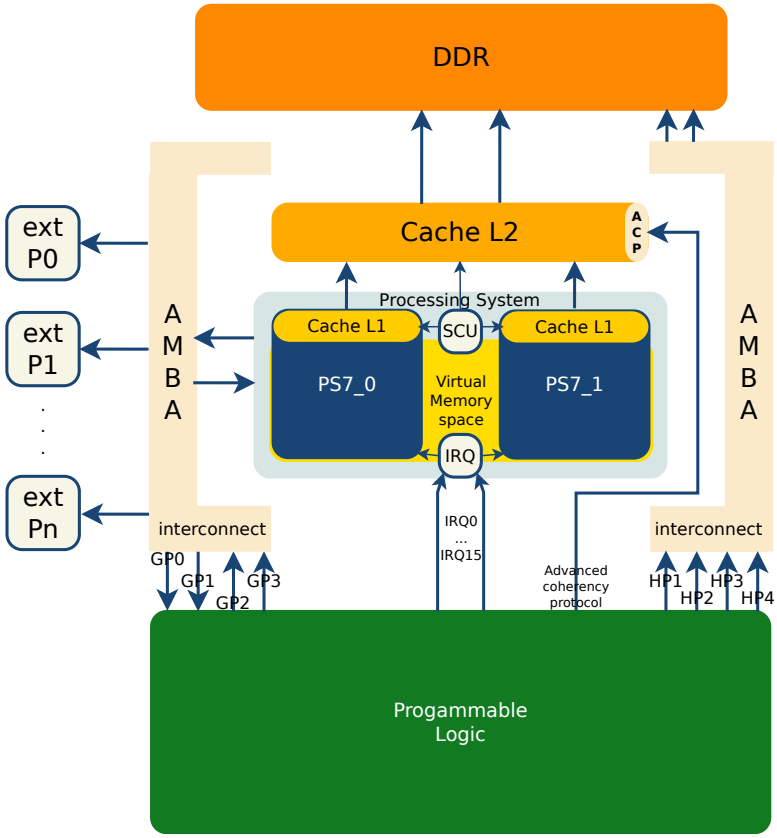


Figure 4-8 – **Overview of the Xilinx Zynq structure.**

2.3 Xilinx Zynq

2.3-1 Zynq structure

The Xilinx Zynq device [Inc16] is a heterogeneous architecture mainly composed of two parts:

1. Programmable System (PS) which is built around two ARM Cortex A9 cores coupled with dedicated peripherals such as SPI, I2C, CAN, UART, GPIO, SDIO, USB and GigEthernet.

2. Programmable Logic (PL) which is an FPGA fabric extracted from the Artix7 or Kintex7 family.

Fig. 4-8 presents an overview of the communication infrastructure of the Zynq. As we can see, a set of interfaces is implemented to tightly link these parts and enable fast communications and synchronization between them. These interfaces are built around the AXI structure and could be divided into two parts:

- General Purpose Port (GP): Four of them are available within the Zynq devices. GP0 and GP1 are master AXI ports belonging to the PS. GP2 and GP3 are master AXI ports belonging to the PL. Each of them are connected to a slave AXI port in the PL for GP0 and GP1, and respectively in the PS for GP2 and GP3.

- Dedicated memory port: there exists four High Performance (HP) ports that enable the PL to directly access data within the DDR memory. Each of them is a full-duplex 64 bit connection, meaning that at every clock cycle, a total of 16 Bytes of data can be transferred on AXI read and AXI write channels concurrently. There is also an Accelerator Coherency Port (ACP) which is connected to the ARM Snoop Control Unit (SCU) and which could therefore initiate cache coherent accesses to the ARM sub-system from the PL.

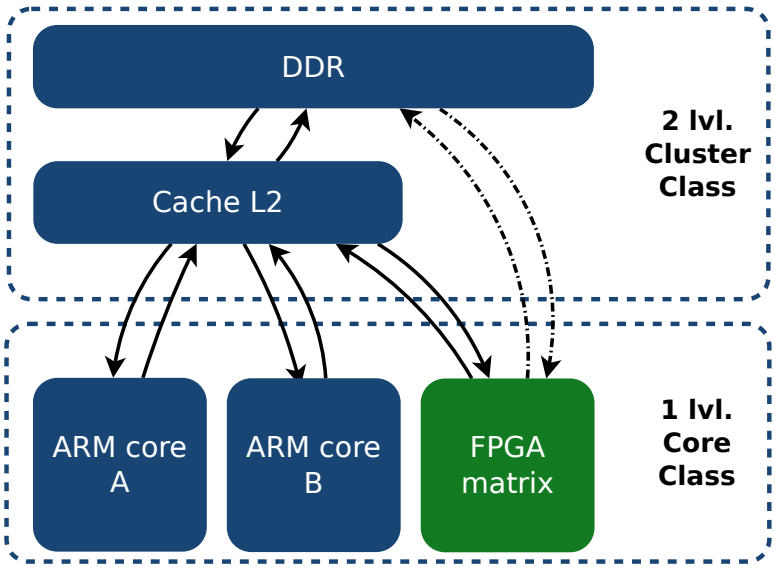


Figure 4-9 – **Details of the Xilinx Zynq memory hierarchy.**

The Zynq device also contains an interruption mechanism between the PL and PS through 16 IRQ lines.

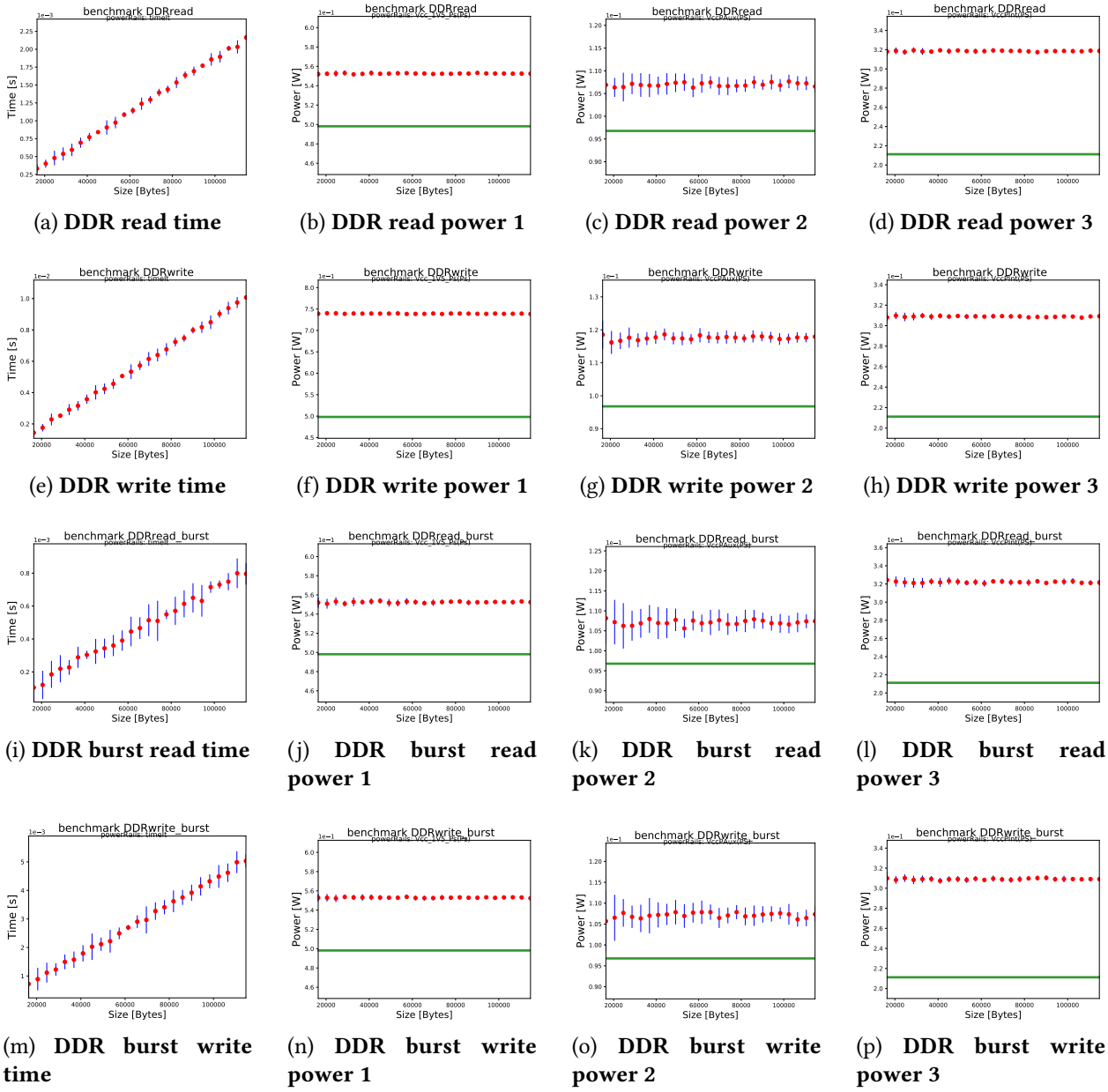
Experiment infrastructure: Experiments are based on the Zynq Zc702 evaluation board. The ARM cores run a Linux operating system *(Linux v4.0.0)*. Power consumption is measured through the *Power Management Bus (PMBus)* embedded on the board. In our experiments, a HW component samples at regular time intervals 7 input power rails of the Zynq SoC. A new sample could be retrieved every 1 ms with 16 bits dedicated for current value and 16 dedicated to voltage. A sample frequency of 142Hz could be achieved, when the 7 rails are monitored at the same time. These measures are used to compute the instantaneous power and the energy variation over the micro-benchmark execution.

2.3-2 Power analysis

Fig. 4-9 shows the memory hierarchy tree for a single Zynq component. The Zynq embedded processors can access to different memory hierarchy levels with various channels, as listed below:

- Access to L2 cache memory:
  - SW channels use standard memory access.
  - HW channels use ACP.
- Access to DDR memory:
  - SW channels use standard memory access.
  - HW channels use HP.

Unmapped memory channels can also be used. Heterogeneous communications occur without memory bank access through GP using two synchronization modes: pooling and ARM *Interrupt ReQuest (IRQ)*. In order to extract the communication cost of these channels, a set of six micro-benchmarks is used, as detailed below:



(a) **DDR read time** (b) **DDR read power 1** (c) **DDR read power 2** (d) **DDR read power 3**

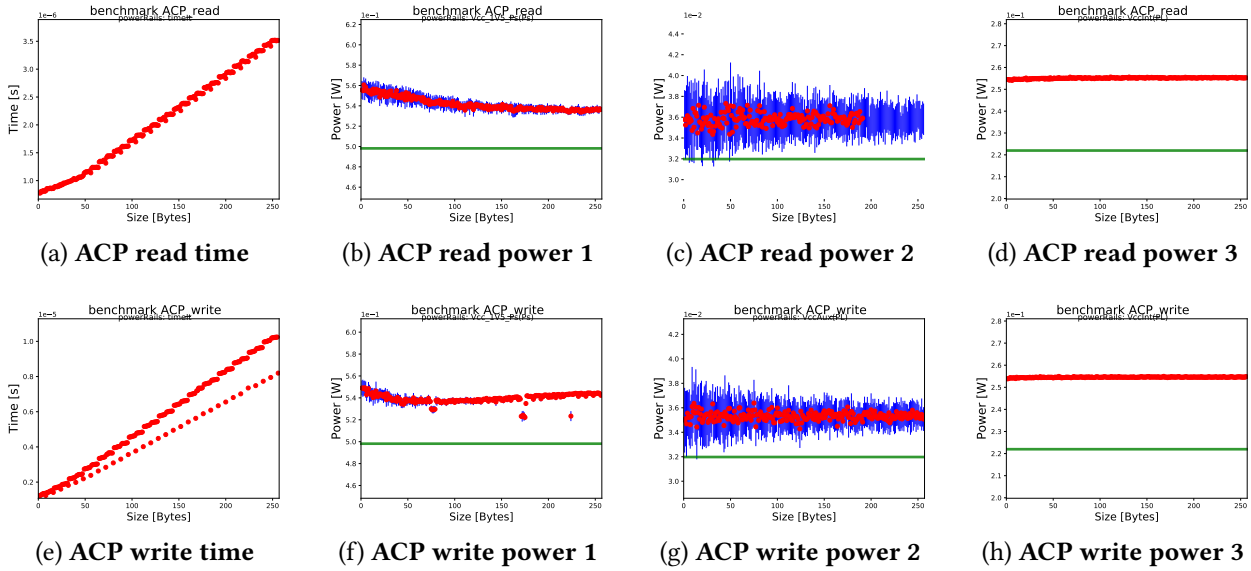
(e) **DDR write time** (f) **DDR write power 1** (g) **DDR write power 2** (h) **DDR write power 3**

(i) **DDR burst read time** (j) **DDR burst read power 1** (k) **DDR burst read power 2** (l) **DDR burst read power 3**

(m) **DDR burst write time** (n) **DDR burst write power 1** (o) **DDR burst write power 2** (p) **DDR burst write power 3**

Figure 4-10 – **Average power measurement and time for the DDR channel (only the relevant rails appear). The variations between the** 20 **launches are depicted through the blue interval around the red average points.**

- CL1: for channel between SW cores and the first memory level. It generates read or write operations from SW core in an array located in the L1 cache.
- CL2: for channel between SW cores and the second memory level. It generates read or write operations from SW core in an array located in the L2 cache.
- DDR: for channel between SW cores and the external memory. It generates read or write operations from SW core in an array located in the DDR.
- HPx: for channel between HW core and the external memory. It generates read and write operations from HW core in an array located in the DDR.
- ACP: for channel between HW core and the second memory level. It generates read and write operations from HW core in an array located in the L2 cache.



(a) **ACP read time** (b) **ACP read power 1** (c) **ACP read power 2** (d) **ACP read power 3**

(e) **ACP write time** (f) **ACP write power 1** (g) **ACP write power 2** (h) **ACP write power 3**

Figure 4-11 – **Average power measurement and time for the ACP memory channel (only the relevant rails appear). The variations between the 20 launches are depicted through the blue interval around the red average points.**

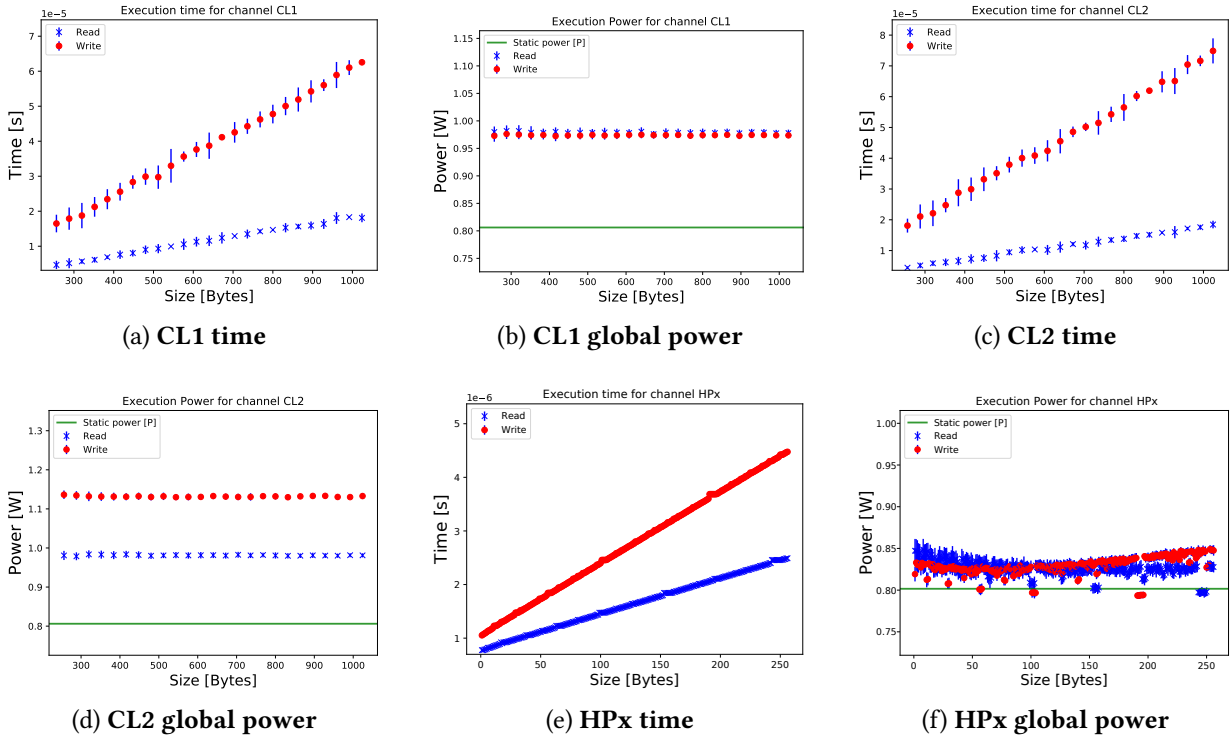
- GPx: for unmapped communications on GP. It generates ping-pong control flow between SW and HW with and without IRQ enabled.

Each micro-benchmark family listed above, except GPx, is executed on two configurations to illustrate the memory line phenomenon. The first configuration generates cache misses on each memory access, while the second one retrieves a full memory line between two misses, which highlights the burst access in memory.

Fig. 4-10 and Fig 4-11 show the execution time and power of DDR and ACP micro-benchmark for multiples communications sizes. The power measurements depict only the relevant power rails, for which the micro-benchmark induces a variation over the static consumption (green line curves). Fig. 4-10 targets the DDR communications channels, the micro-benchmarks are launched 20 times. The variations between the 20 launches are depicted through the blue interval around the red average points. For the software communication channels, the time is measured with linux managed counter. The hardware micro-benchmarks use a hardware counter implemented within the PL. Fig. 4-11 presents results on the ACP communication channel. The maximum burst size of this channel is well defined in the Zynq documentations and limited to 256 Bytes. This is why the measurement range is limited compared to the DDR channels. As we can see on Fig. 4-11e, and in a lesser extent on Fig. 4-11a, there is a phenomenon linked to a bus alignment effect. We believe that this phenomenon is due to the presence of a fast path for the 8 Bytes aligned read and write accesses. It introduces a bias that makes aligned and full access faster than smaller incomplete one over ACP channel.

Fig.4-12 displays a concatenation view of the previous graph for channels CL1, CL2, and HP. All these results are then merged for each channel in order to obtain the energy consumption following the size of the communication. Fig. 4-13 shows the energy curves for each communication channel. As expected, these curves fit well with a second order function. As explained before, the effect of incomplete access over ACP is also visible on Fig. 4-13d.

Two extra micro-benchmarks are used to measure the communication cost at a coarser grain, since the previous set cannot consider communication side effects such as synchronizations. These micro-benchmarks highlight synchronization impact on power consumption through two commu-



(a) **CL1 time** (b) **CL1 global power** (c) **CL2 time**

(d) **CL2 global power** (e) **HPx time** (f) **HPx global power**

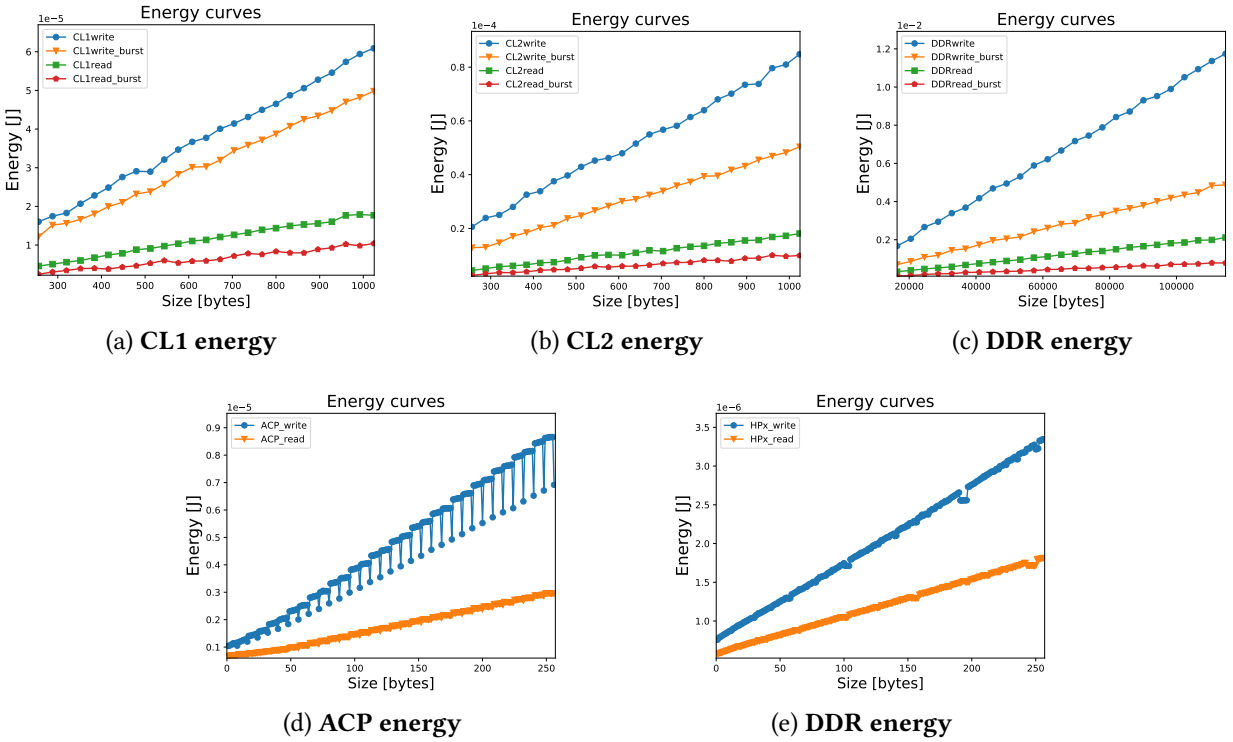
Figure 4-12 – **Time and power summary for various channels.**

nications patterns, named *pipeline* and *shared*. They are illustrated on Fig. 4-14 and explained thereafter.

Pipeline pattern When a set of BpBs requires to process the same input sequentially, the *pipeline* communication pattern is involved. The micro-benchmark uses two BpBs to illustrate this pattern (Fig. 4-14a). The first block consumes data from the input channel and produces output for the second one. The second block reads this data and produces results that are sent to the output channel. Fig. 4-15 displays the measurement curves obtained for this communication pattern.

Shared pattern This pattern appears in data-parallel applications. The micro-benchmark is composed of two BpBs working on the same input data block (Fig. 4-14b). Theses two blocks read part of input data and produce their own data chunk that will be updated in the input block. Before updating the input block, BpBs are synchronized with a barrier, which is a synchronization mechanism where multiple BpBs can wait for (as opposed to point-to-point synchronization mechanism). Fig. 4-16 displays the measurement curves obtained for this communication pattern.

The results of the previous experiments performed on the Zynq show that the variance of the measurements is quite small. The power consumption is nearly constant during the execution and the execution time could therefore be approximated with a linear function. In the following, the execution time and the energy cost of communication are approximated as a linear function $f(bytes) = a \times bytes + b$, where the values a and b are respectively the dynamic and static parts. Table 4-2 gives the execution time and energy cost parameters extracted on the Zynq architecture with the micro-benchmark set detailed before. These coefficients will be used in (4-5) and (4-6) to compute the energy cost of communications. The validity domains of these functions are ranging

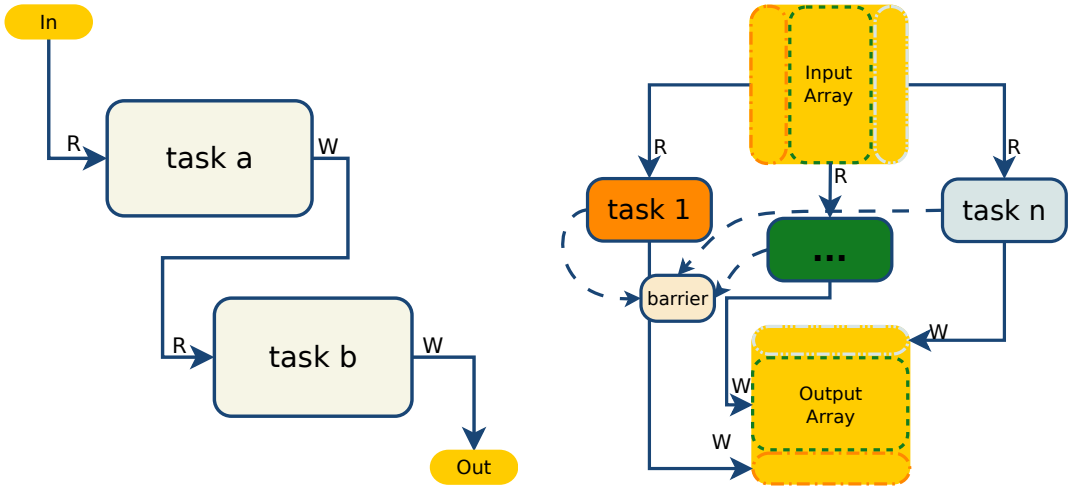


(a) **CL1 energy** (b) **CL2 energy** (c) **DDR energy**

(d) **ACP energy** (e) **DDR energy**

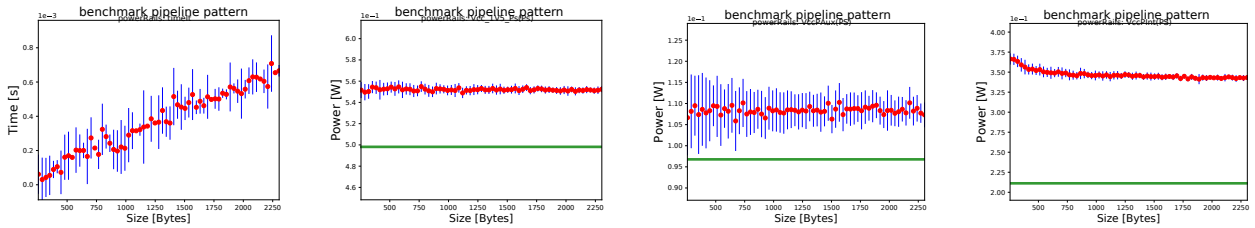
Figure 4-13 – **Average energy consumed on each communication channel following the communication size.**

from [128 bytes, 1 Kbytes] for the L1 cache memory channel to [4 Kbytes, 128 Kbytes] for the DDR memory channel.



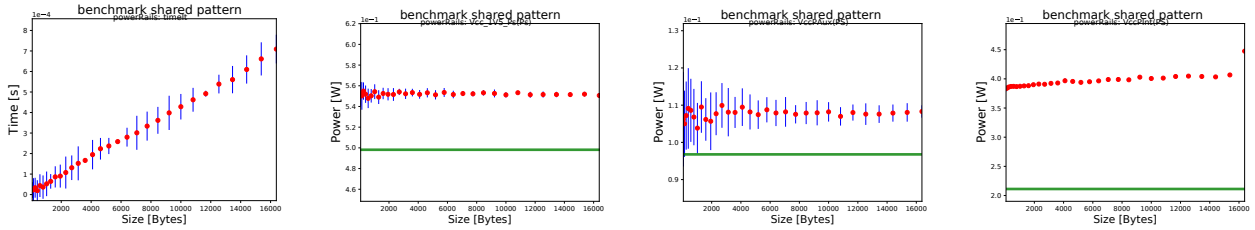
(a) **Pipeline manner.** (b) **Shared manner.**

Figure 4-14 – **Communication patterns.**



(a) **Pipeline time** (b) **Pipeline power 1** (c) **Pipeline power 2** (d) **Pipeline power 3**

Figure 4-15 – **Average power measurement and time for the Pipeline communication pattern (only the relevant rails appear).**



(a) **Shared time** (b) **Shared power 1** (c) **Shared power 2** (d) **Shared power 3**

Figure 4-16 – **Average power measurement and time for the Shared communication pattern (only the relevant rails appear).**

Benchmark	Time [s] $f : x \rightarrow ax + b$		Energy [J] $f : x \rightarrow ax + b$	
	a	b	a	b
HPx read	6.71e-09	7.82e-07	5.56e-11	6.49e-09
HPx write	1.34e-08	1.06e-06	1.18e-10	9.37e-09
ACP read	1.14e-08	6.07e-07	9.97e-11	5.30e-09
ACP write	3.59e-08	8.97e-07	2.78e-10	6.95e-09
GPx polling	5.41e-07	0	8.27e-09	0
GPx irq	2.85e-06	0	9.47e-08	0
DDR read	1.86e-08	7.48e-06	1.54e-09	6.16e-07
DDR read burst	7.06e-09	1.54e-06	6.07e-10	1.32e-07
DDR write	8.76e-08	-3.84e-06	2.37e-08	-1.04e-06
DDR write burst	4.40e-08	-1.34e-05	3.24e-09	-9.85e-07
CL1 read	1.82e-08	-2.95e-08	1.52e-09	-2.45e-09
CL1 read burst	1.02e-08	6.68e-09	8.40e-10	5.50e-10
CL1 write	6.03e-08	3.12e-07	4.72e-09	2.44e-08
CL1 write burst	5.05e-08	-3.73e-07	3.73e-09	-2.75e-08
CL2 read	1.76e-08	-2.69e-08	1.51e-09	-2.30e-09
CL2 read burst	9.62e-09	3.19e-07	8.01e-10	2.66e-08
CL2 write	7.19e-08	-5.46e-09	1.70e-08	-1.29e-09
CL2 write burst	5.08e-08	-4.14e-07	3.71e-09	-3.02e-08
Shared Pattern CL1	4.15e-08	1.54e-05	6.66e-09	2.46e-06
Shared Pattern CL2	4.95e-08	-4.79e-04	1.54e-08	-1.49e-04
Shared Pattern DDR	6.08e-08	-7.41e-03	2.87e-08	-3.49e-03
Pipeline pattern	2.98e-07	-1.03e-05	3.32e-08	-1.15e-06
Static Power	n.v.	n.v.	1.20e+00	n.v.

Table 4-2 – **Extracted power model parameters for the Zynq architecture.**

3 Power model validation on Xilinx zynq

This section validates the communication-based power model introduced in Section 1 on a set of random communication-centric applications called mutant. First, we define the concept of the mutant application and we explain their general structure. Then in Sub-Section 3.2, we show the communication-based energy estimation results versus the measured one on 80 mutant applications to validate the proposed power modeling approach.

3.1 The mutant application principle

In this subsection, *mutant applications* are generated to validate the parameter values extracted from previous micro-benchmark characterization. A mutant application is an abstract application automatically generated from pattern functions. It randomly generates communication traffic over different communication channels of the target architecture. To this purpose, 12 SW and 6 HW functions were written for the Zynq architecture. Each of them generates communication traffic on a communication channel following a specific mode. These function patterns contain no computation operation, so the computation energy cost of each one can be neglected $Ecomp = 0$. Then, a mutant generator framework combines these functions randomly to obtain an application that stresses the overall communication channels at the same time. The generator framework are described through the following Algorithm 2. Fig. 4-17 depicts the structure of mutant applications, which are composed of several rounds.

At the beginning the generator arranges pattern functions in an array and allocates memory for the communication channel and configuration values. Then, the power measurement starts and the generator begins to iterate on the targeted number of rounds. For each iteration, the generator chooses randomly two SW functions and one HW function. For each function, a communication size is randomly chosen within the validity range of the involved communication channel. Once functions and communication size are chosen, the generator generates a thread for each function and waits for its ending. Then, the configuration step is logged. When all rounds are finished, the generator stops power measurement and writes configuration logs and power measurement values into files.

3.2 Mutant validation

For the validation purpose, 80 mutants have been generated. Each of them was executed on the evaluation board and the energy-consumption was compared to the output of the communication-based power model computed before.

Table 4-4 shows the configuration of four generated mutants applications. Their configuration is summarized by the overall generated communications. The percentages of communications over the different channel are also displayed.

mutantRank	Time [s]		Energy [J]		Error	
	measured	estimated	measured	estimated	time	energy
mutant 1	2.308	2.311	2.949	2.943	0.1%	0.2%
mutant 2	2.340	2.336	3.031	2.964	0.2%	2.2%
mutant 3	2.775	2.780	3.621	3.540	0.2%	2.3%
mutant 4	2.828	2.833	3.739	3.624	0.2%	3.1%
average on 80 mutants	2.974	2.975	3.855	3.861	0.5 %	0.2 %

Table 4-3 – **Power estimation results of mutant executions.**

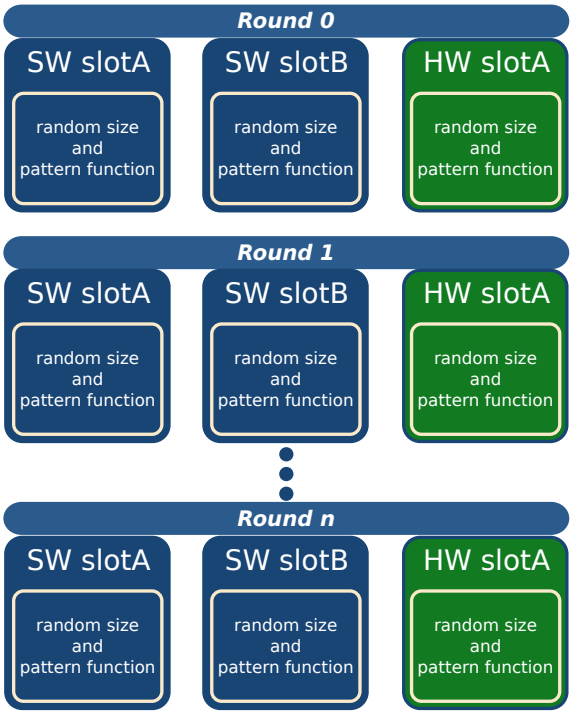


Figure 4-17 – **Overview of a mutant application structure.**

Data: Number of rounds
swFunc = getSwFunctionList()
hwFunc = getSwFunctionList()
AllocComChannelMemory()
AllocLoggingMemory()
startPowerMeasure()
for *round* ***in*** *number of rounds* **do**
  swA = randomConfig()
  swB = randomConfig()
  hw = randomConfig()
  workerA = spawnSwThread(swFunc, swA)
  workerB = spawnSwThread(swFunc, swB)
  workerH = spawnSwThread(hwFunc, hw)
  waitThread(workerA, workerB, workerH)
  log[round] = logconfig(SwA, sWB, hw)
end
stopPowerMeasure()
for *round* ***in*** *number of rounds* **do**
  writeConfigToFile(log[round])
end
writePowerMeasureToFile()
Algorithm 2: Mutant generator framework.

Table 4-3 details the results of the power consumption estimation on the four described mutant applications. The last line shows average results obtained over 80 mutants. The estimated values of Table 4-3 were computed with parameters shown in Table 4-2 and the mutant configuration. These inputs were used in the communication-based power model presented in Section 1. The mutant configuration contains the amount of communication between each pair of blocks and was used alongside the architecture parameters (cf. Table 4-2 to compute the communication energy cost and time. Then the mutant critical path was computed, as the sum of the maximum execution time of each round, and associated with static power parameter to compute the static power cost. Table 4-3 also gives the error between measured and estimated values of the mutant application power consumption. The errors obtained for the previous set of *mutant applications* are lower than 3.1%, which is mainly due to the measurement infrastructure inaccuracy. The three first lines show that the communication channel division has no incidence on the power estimation error. These experiments validate the parameter values obtain through the micro-benchmarking method and the communication-based power model.

The power estimation computed with a mono-threaded python script on *Intel i5 Haswell-ult* processor takes 0.55 second on the 80 previous mutants. The obtained accuracy and computation time of the introduced communication-based power model shows that our model seems to be an

efficient approach to target task mapping issues with low complexity.

4 Conclusion

This chapter introduced a new power modelling approach focusing on communication channels. This power model targets generic heterogeneous multicore architectures. The proposed power model is mainly communication-centric and aims at simplifying the task mapping step under energy or power constraints. A micro-benchmarking approach was introduced to enable the identification of the target architecture parameters defined in the power model. This identification method was experimented on Kalray MPPA and on the Xilinx Zynq architecture. Due to the high power measurement inaccuracy of MPPA measurement tool, the method was only validated on the Zynq architecture. The combination of communication-based power model and parameter estimation method based on micro-benchmarking has shown its efficiency on numerous synthetic applications. The achieved estimation accuracy is largely enough for being used in the task mapping step, while the estimation time also fits design space exploration constraints.

mutantRank	Total bytes	Channel name													
		Cache L1		Cache L2		DDR		HPx		ACP		GPx	
mutant 1	4.56e+07	read	6.6%	read	1.3%	read	1.3%	read	1.2%	read	0.6%	polling	3.5%
		read burst	0.6%	read burst	5.5%	read burst	18.0%						
		write	6.8%	write	2.5%	write	1.1%	write	2.0%	write	0.4%	irq	0.5%
		write burst	6.6%	write burst	0.0%	write burst	41.3%						
mutant 2	5.37e+07	read	6.7%	read	2.1%	read	4.7%	read	2.9%	read	4.8%	polling	2.0%
		read burst	4.7%	read burst	0.2%	read burst	10.0%						
		write	5.0%	write	0.6%	write	0.0%	write	7.4%	write	2.0%	irq	0.5%
		write burst	4.7%	write burst	6.8%	write burst	35.0%						
mutant 3	4.10e+07	read	5.3%	read	0.0%	read	5.0%	read	3.3%	read	0.0%	polling	3.0%
		read burst	1.9%	read burst	0.0%	read burst	11.7%						
		write	7.6%	write	6.1%	write	0.6%	write	1.9%	write	8.7%	irq	1.2%
		write burst	7.2%	write burst	3.9%	write burst	32.6%						
mutant 4	4.21e+07	read	4.1%	read	2.2%	read	2.9%	read	5.0%	read	0.1%	polling	1.1%
		read burst	6.1%	read burst	1.8%	read burst	13.9%						
		write	16.3%	write	0.4%	write	7.2%	write	5.5%	write	3.4%	irq	1.4%
		write burst	1.6%	write burst	2.1%	write burst	25.2%						

Table 4-4 – **Communications involved in mutant applications.**

Chapter **FIVE**

ENERGY-DRIVEN ACCELERATOR EXPLORATION FOR HMPSOC

Contents

This chapter proposes and validates an exploration method for partitioning applications on software cores and hardware accelerators under energy-efficiency constraints. The methodology is based on energy and performance measurements of a tiny subset of the design space and an analytical formulation of the performance and energy of an application kernel mapped on a heterogeneous architecture. A tiled-DSE method based on the analytical power model proposed in the previous chapter is introduced to circumvent the computation time bottleneck of state-of-the-art power models. The execution costs of tasks are directly extracted on the real architecture and inserted in the communication-based power model. The proposed method mainly focuses on accelerator building, considering only the acceleration of the critical tasks that generate bottlenecks in the application. For this purpose, we formulate the problem with Mixed Integer Linear Programming (MILP) and solve it within less than a second. The approach is validated on two application kernels (matrix multiply and stencil) using Zynq-based architecture showing more than 12 % acceleration speed-up and energy saving compared to standard approaches. Results also show that the most energy-efficient solution is application- and platform-dependent and moreover hardly predictable, which highlights the need for fast exploration.

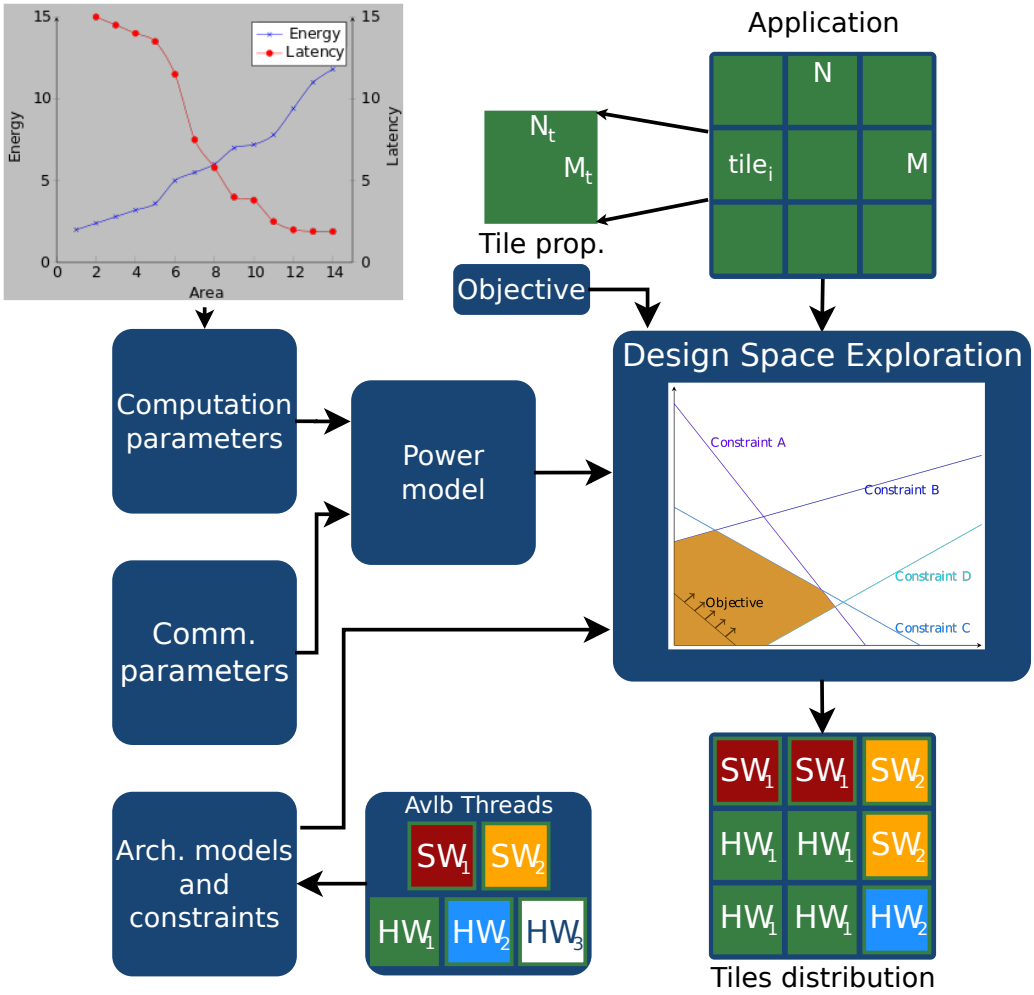


Figure 5-1 – **Overview of the proposed DSE.**

The chapter is organized as follows. Section 1 introduces an overview of the proposed tiled-DSE method. Section 2 presents the extraction method of the time and energy parameters of the computation tasks. Section 3 analyzes the design space size and proposes the MILP formulation as a set of constraints. Experimental setup is given in Section 4. Section 5 shows the parameter extraction results and evaluates the resulting configuration of the proposed DSE. Finally, conclusions are given in Section 6.

1 Overview of Proposed Tiled-DSE Flow

This section introduces the proposed tiled design flow and briefly describes the content of each design step. The proposed tiled-DSE, depicted in Fig. 5-1, proposes to optimize the HW/SW partitioning and mapping under user-defined objectives, especially an energy constraint. The flow targets tiling-based parallel applications and relies on an analytical power model that provides the tiled-DSE framework with the execution time and energy of a HW/SW configuration.

1.1 Tiling-based parallel applications

Tiling is a well-known parallelization technique in the compiler domain [WL91]. In general, tiling maps an n-deep loop nest into a $2n$-deep one where the most inner n loops include only a

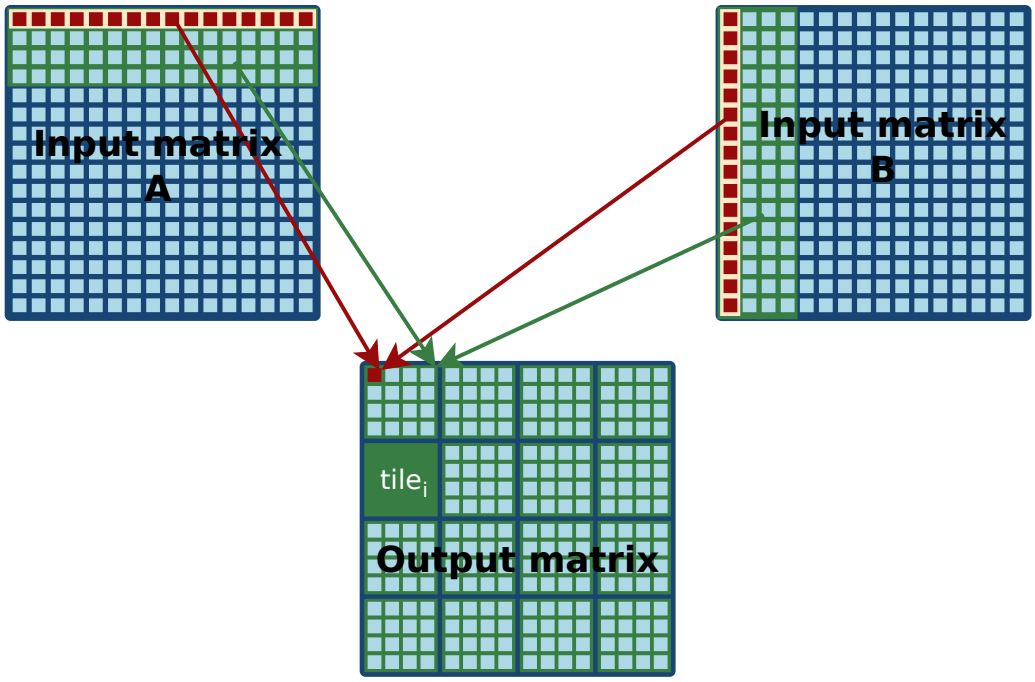


Figure 5-2 – **Matmult tiling structure.**

fixed number of iterations. Tiling refers to the partitioning of the iteration space into polyhedral blocks. This transformation was initially introduced to increase data locality of applications. Tiling reduces the volume of data accessed between reuses of an element, allowing a reusable element to remain in the cache until the next time it is accessed. When dealing with accelerator, this property is useful as it comes with the use of small scratchpad memory with efficient data-reuse. After preliminary transformations on the iteration space (such as skewing, loop-merging, etc.), tiling exposes the data parallelism of an application. As a consequence, within a dimension of the iteration space, tiles are independent and could be executed in parallel. In short, the tiling transformation has three main advantages in the case of accelerator design: it increases data-reuse with small scratchpad buffer; it exposes coarse-grain parallelism that could be exploited in HW/SW partitioning; within each tile, fine-grain parallelism could be used to draw the full power of the HW architecture. As an example, PARSEC benchmarks [Bie+08], considered as a representative batch of applications, include 8 applications over the 12 available which are data-parallel. Thus, after small transformations on the iteration space, we can consider that $2/3$ of applications in most of the benchmark suites are good candidates for tiling.

Fig. 5-2 depicts the tiling step for a matrix multiply application kernel based on $N \times N$ matrices. In matrix multiply one output value depends on one row and one column in the input matrices (depicted in red). We want to apply a basic square tiling scheme of size $t \times t$. The tiling scheme is applied on the output matrix in order to split the computation in $N_{tiles} = \frac{N}{t}$ independent bunches of computation. As we can see, this decomposition implies that each output tile reads a bunch of t rows in the input matrix A and a bunch of t columns in the input matrix B (depicted in green). In the following, we focus on 2D-applications with an N by M iteration space. Tiling scheme is applied on the output iteration spaces with rectangular tiles composed of N_t by M_t elements. The inputs read scheme is driven by the target application data-dependency scheme. Thus, a set of $N_{tiles} = \frac{N}{N_t} \times \frac{M}{M_t}$ tiles are distributed among the execution cores of the target architecture. The tiled application is depicted on top of the tiled-DSE flow in Fig. 5-1. Allocation of tiles to HW and SW resources is the output of the tiled-DSE flow.

1.2 Heterogeneous Architectures

In our model, heterogeneous architectures are considered to be composed of N_{SW} software and N_{HW} hardware computation cores. All SW cores could rely on the same processor architecture or could be a set of heterogeneous processor cores. HW cores are implemented in the Programmable Logic (PL) fabric. The PL can be partitioned into at most N_{HW} independent HW cores where N_{HW} is the number of available memory ports within the PL. This restriction, based on the maximum number of simultaneous data transfers to/from the main memory, especially prevents the occurrence of memory congestion that leads to unpredictable memory access-time and energy cost. Moreover, all the resources used by the cores should be less than the overall resources available in the architecture. For instance, the Zynq architecture from Xilinx is a representative case of heterogeneous architectures. This architecture is composed of two homogeneous SW cores tightly coupled with an FPGA PL fabric. HW cores communicate with SW ones through the four HP connected to the DDR memory (cf. Chapter 4 2.3. In this chapter, our experiments will be based on the Zynq architecture.

1.3 Tiled-DSE Objectives

The goal of the proposed tiled-DSE method is to find the best configuration that minimizes the user objective (e.g. execution time or total energy consumption of the application). A configuration is a vector $\vec{C}$ composed of the tile distribution among the SW and HW cores, the SW core type used for each of the N_{SW} SW core, and the implementation type used for each of the N_{HW} cores. An example of such distribution is represented in Fig. 5-1 at the output of the tiled-DSE. Let $tiles^i_{SW}$ and $tiles^j_{HW}$ be the number of tiles allocated to the SW core i and to HW core j, respectively. The SW cores could use one type within a set of available Instruction Set Architecture (ISA), the HW cores could use one computation block within a set of available implementations which have different area-latency-energy trade-offs. As an example, Fig. 5-1 depicts on its upper left different HW designs, each of them corresponding to an implementation of a computation block with different area, energy, and latency costs. For generality purpose, such implementations can be obtained through hardware exploration tools such as [Sha+15]. In our case, hardware exploration was manually performed during the experiments with the help of high-level synthesis tool. The computation block specification is synthesized under various latency constraints to obtain different implementations varying from 1 to N_{impl}, with $type^j_{HW}$ representing the implementation used in the HW core j. The available SW core type could vary from 1 to N_{ISA}, with $type^i_{SW}$ representing the ISA used for SW core i. Under these assumptions, a configuration $\vec{C}$ is defined as:

$$\vec{C} = \left[(tiles^1_{SW}, type^1_{SW}); \ldots; (tiles^{N_{SW}}_{SW}, type^{N_{SW}}_{SW}); (tiles^1_{HW}, type^1_{HW}); \ldots; (tiles^{N_{HW}}_{HW}, type^{N_{HW}}_{HW})\right], \tag{5-1}$$

where the tuple $(tiles^i_{SW}, type^i_{SW})$ is the number of tiles allocated to SW core i using ISA $type^i_{SW}$, and the tuple $(tiles^j_{HW}, type_j)$ is the number of tiles allocated to HW core j using a computation block build upon implementation $type^j_{HW}$. The aim of the tiled-DSE is therefore to find the optimal configuration $\vec{C_{opt}}$ that minimizes the cost (time, energy) objective such as:

$$\vec{C_{opt}} = \min_{\forall \vec{C}} Cost(\vec{C}), \tag{5-2}$$

where the function $Cost(\bullet)$ relies on the analytical models described in the next section.

1.4 Energy and execution time models

The execution time and power models are introduced in this section. The energy consumption of an application executed on a heterogeneous multiprocessor architecture depends on three main sources: the static energy dissipated during execution time, the dynamic energy consumption used for computations, and the energy used for communications between processing cores. Applications are composed of N_{tiles} independent tiles that could be executed in parallel. The execution of a tile is atomic and the energy and time needed for its computation only depends on the targeted execution core. The amount of required data for tile computation is assumed to be known at compile time. HW cores are managed with low-level calls and interruptions which are encapsulated within a SW thread. Spawning the threads among computation cores is sequentially performed and begins with HW cores to minimize the number of context switch within the operating system.

1.4-1 Computation time

For a configuration $\vec{C}$, the total computation time is:

$$T_t(\vec{C}) = \max\left[T_{HW}(\vec{C}), T_{SW}(\vec{C})\right], \tag{5-3}$$

where T_{SW} and T_{HW} corresponds to the computation time on the SW and HW cores for a given configuration, respectively. T_{SW} and T_{HW} are computed as:

$$\begin{aligned} T_{HW}(\vec{C}) &= \max_{j\in(1\ldots N_{HW})}\left[T_{comp_{HW}}^{type_{HW}^{j}} \times tiles_{HW}^{j} + j \times T_{spawn}\right] \\ T_{SW}(\vec{C}) &= \max_{i\in(1\ldots N_{SW})}\left[T_{comp_{SW}}^{type_{SW}^{i}} \times tiles_{SW}^{i} + (N_{HW}^{used}(\vec{C}) + i) \times T_{spawn}\right] \end{aligned} \tag{5-4}$$

with $T_{comp_{HW}}^{type_{HW}^{j}}$ and $T_{comp_{SW}}^{type_{SW}^{i}}$ the computation time of a tile computed on HW implementation $type_{HW}^{j}$ and on SW ISA $type_{SW}^{i}$, respectively. T_{spawn} represents the time needed to configure and spawn a computation thread. $N_{HW}^{used}(\vec{C})$ represents the number of HW cores used with configuration $\vec{C}$:

$$N_{HW}^{used}(\vec{C}) = \sum_{j=1}^{N_{HW}} \begin{cases} 1, & \text{if } tiles_{HW}^{j} > 0 \\ 0, & \text{else} \end{cases} . \tag{5-5}$$

1.4-2 Energy consumption

The total energy $E_t(\vec{C})$ consumed by the execution of the tiled application on configuration $\vec{C}$ comprises static energy and dynamic energy for both computations and communications:

$$E_t(\vec{C}) = E_{stat} + \sum_{i=1}^{N_{SW}}(E_{comp_{SW}}^{type_{SW}^{i}} + E_{com_{SW}}) \times tiles_{SW}^{i} + \sum_{j=1}^{N_{HW}}(E_{comp_{HW}}^{type_{HW}^{j}} + E_{com_{HW}}) \times tiles_{HW}^{j}, \tag{5-6}$$

where E_{stat} is the static energy consumption, $E_{com_{SW}}$ and $E_{com_{HW}}$ are the energy of communications for a tile computed on SW and HW, respectively. $E_{comp_{SW}}^{type_{SW}^{i}}$ and $E_{comp_{HW}}^{type_{HW}^{j}}$ represent the energy required to compute a tile on SW core i with ISA $type_{SW}^{i}$ and on HW implementation $type_{HW}^{j}$, respectively. In the following, we propose a method to estimate the computational energy of a tile, while the energy due to communication between cores can be estimated with the method based on micro-benchmarking described in the previous chapter.

2 Computation Parameter Extraction

Computation parameters represent the energy and time required to execute a tile for a given implementation, *i.e.*, HW implementation type or ISA SW core type. To extract these parameters, execution traces with time and energy information are required for each configuration. To ease the extraction process, only a subset of the design space built around a basic architecture composed of one SW core and one HW core is considered. Measurements traces are obtained by varying the tile distribution in this architecture subset. Therefore, with an application on N_{tiles} tiles, an architecture with N_{impl} HW implementations and N_{ISA} types of SW processor and assuming that $N_{tiles} >= 0$, $N_{impl} > 0$, and $N_{ISA} > 0$, the subset size is defined by the following polyhedron:

$$[N_{tiles}, N_{impl}, N_{ISA}] \rightarrow (1 + N_{tiles}).\max(N_{impl}, N_{ISA}). \tag{5-7}$$

In addition, we vary the tile distribution between HW and SW with a coarse grain scale, which leads to diminish the number of execution traces needed.

As an example, if we consider a target architecture that could be composed of up to two SW cores alongside with up to four HW cores, a tiled target application with $N_{tiles} = 256$, four available HW implementations for HW cores and two available ISA for SW cores, we obtain a global design space exploration composed of more than 1×10^{12} configurations. By limiting the execution traces to a subset design space composed of basic architectures, the design space is reduced to 1028 configurations. On top of that, the tile distribution variation at coarse grain scale reduces the number of solutions needed to be measured to 36 configurations, which is quiet negligible compared to the whole design space. The 36 obtain execution traces are then processed with a Least Squared Root (LSR) algorithm to extract parameters. This algorithm computes the parameter values that minimize the squared error between the measurements and the cost function.

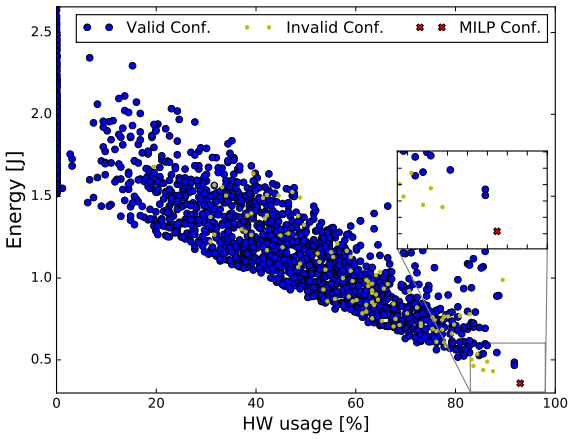
In Sub-Section 5.1, we show that, only 9 tile distributions are required to achieve a good parameter extraction accuracy with an application composed of $N_{tiles} = 256$ tiles. The basic architecture approach combined with the coarse grain tile distribution variation can extract the computation parameters with a very small subset of execution traces in the total design space. In the following, we call these 9 coarse grain tile distributions as sample configurations.

The execution time parameters are extracted with the time cost function defined in (5-4). For each execution trace, the sample configuration involving implementation $type^j_{HW}$ and SW ISA $type^i_{SW}$ is known as well as the overall execution time. The LSR algorithm extracts the values of parameters $T^{type^j_{HW}}_{comp_{HW}}$, $T^{type^i_{SW}}_{comp_{SW}}$ and T_{spawn}.

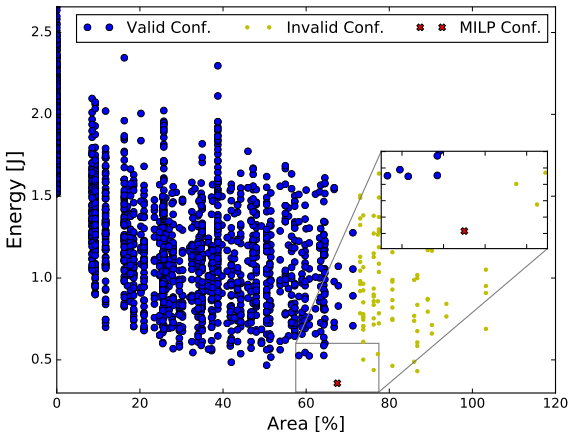
The energy parameters are extracted with the same process. The extraction is based on the energy cost function introduced in (5-6). For an execution trace involving implementation $type^j_{HW}$ and SW ISA $type^i_{SW}$, the overall energy consumption is computed as well as the communication energy, $E_{com_{SW}}$ and $E_{com_{HW}}$ and the static energy E_{stat}. Then, the LSR algorithm extracts the values of parameters $E^{type^j_{HW}}_{comp_{HW}}$ and $E^{type^i_{SW}}_{comp_{SW}}$.

3 Design Space Exploration of Tiled Applications

The second main part of the methodology is the design space exploration itself. In order to demonstrate that exploring the design space would lead to large variation in the cost function and

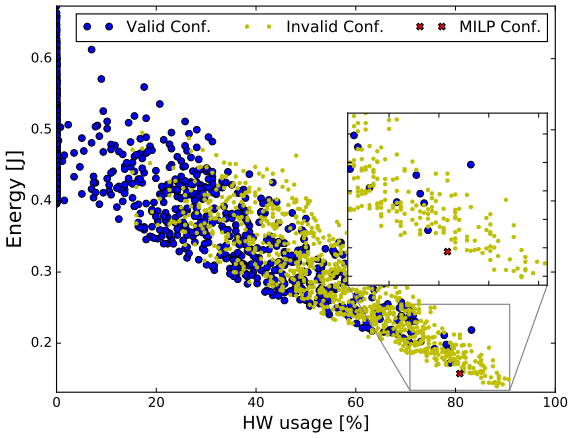


(a) **Matmult energy vs HW use**

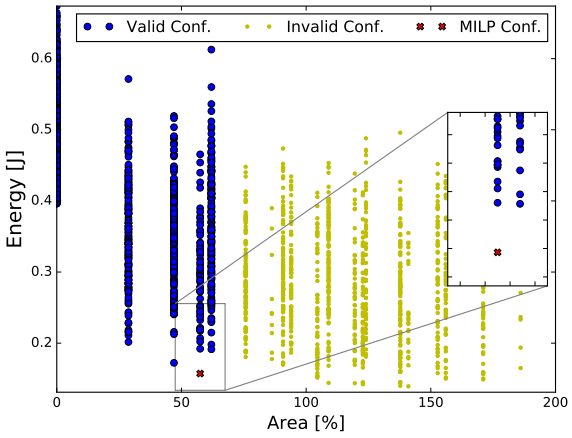


(b) **Matmult energy vs surface**

Figure 5-3 – **Matmult design space exploration.**



(a) **Stencil energy vs HW use**



(b) **Stencil energy vs surface**

Figure 5-4 – **Stencil design space exploration.**

to a prohibitive number of solutions, we first describe the case of an exhaustive search. Then, an MILP formulation is proposed to solve the tiled-DSE problem and find the best solution.

3.1 Exhaustive search

Let the target architecture be similar to the Zynq architecture with two SW cores of the same type and four HW cores. The design space is described as a polyhedron and its size can be calculated with the help of the `isl` library [Ver10]. As a result, the design space size is defined with the following polynomial with N_{impl} and N_{tiles} as parameters:

$$\begin{aligned}[N_{impl}, N_{tiles}] \rightarrow \quad N_{impl}^4 \times (\quad & 1 + \tfrac{137}{60} \times N_{tiles} + \tfrac{15}{8} \times N_{tiles}^2 \\ & + \tfrac{17}{24} \times N_{tiles}^3 + \tfrac{1}{8} \times N_{tiles}^4 + \tfrac{1}{120} \times N_{tiles}^5).\end{aligned} \tag{5-8}$$

For instance with $N_{impl} = 3$ and $N_{tiles} = 256$, the design space is composed of 7.866×10^{11} points. If the energy consumption and execution time of one configuration can be obtained in about 1 ms and using 1 kb of memory, the exhaustive DSE would take more than 9 days and a prohibitive amount of memory, which is therefore not a valid approach. Fig. 5-3a and Fig. 5-4a show a randomly chosen subset of the design space configurations for the matrix multiplication kernel and for the stencil

kernel, respectively (these computation kernels are introduced in the experiment section). These figures plot the energy consumption as a function of the percentage of used HW equivalent to the proportion of tiles computed on the HW accelerator fabric. Fig. 5-3b and Fig. 5-4b show, for the same explored configurations, energy consumption as a function of the percentage of resources used in the HW. These plots show that the design space is very large and that the exploration does not guarantee the implementation feasibility of the obtained configurations. An unfeasible configuration is a configuration that requires more than the available on chip resources (cf. invalid configurations in the previous figures).

To tackle this issue, the following section introduces an optimization method based on an MILP formulation, which solves the problem defined in Eq. 5-2) and finds the best solution in the design space. Fig. 5-3 and Fig. 5-4 also plot the solution obtained thanks to MILP optimization (configuration plotted with a red ×).

3.2 MILP formulation

Mixed-Integer Linear Programming (MILP) is a well-known general framework for solving partitioning problems. With this approach, constraints are defined as a set of inequalities with Boolean, integer (discrete) and non-integer (continuous) variables. Then, solutions can be efficiently determined using commercial or open-source solvers. The optimization is defined with a linear objective function. The intersection of the inequality constraints represents a polyhedron of the feasible solution. The objective function defines a direction into the solution space and the optimal solution is found at the intersection between the objective function and the feasible solutions. This section introduces the set of inequalities that formalizes the constraints of our problem. Then the cost and objective functions are defined.

3.2-1 Model constraints

Coverage constraint To ensure that each tile is computed only once, the coverage constraint is:

$$N_{tiles} = \sum_{i=1}^{N_{SW}} tiles_{SW}^{i} + \sum_{j=1}^{N_{HW}} tiles_{HW}^{j}. \tag{5-9}$$

Unicity constraints To ensure that only one ISA type is used for each SW thread and only one hardware implementation is used for each HW thread, the variables $usedSW_{type}^{i}$ and respectively $usedHW_{type}^{j}$ are defined as:

$$usedSW_{type}^{i} = \begin{cases} 1, & \text{if } type_{SW}^{i} = type \\ 0, & \text{else} \end{cases} \tag{5-10}$$

$$usedHW_{type}^{j} = \begin{cases} 1, & \text{if } type_{HW}^{j} = type \\ 0, & \text{else} \end{cases} \tag{5-11}$$

These variables are binary and can be multiplied by other variables without introducing unsolvable non-linearity in the model. The unicity constraint, ensuring that only one ISA is used in each SW core, is:

$$\forall i \in (1 \dots N_{sW}) : \sum_{type=1}^{N_{ISA}} usedSW_{type}^{i} \leq 1. \tag{5-12}$$

And the unicity constraint, ensuring that only one implementation is used in each HW core, is:

$$\forall j \in (1 \dots N_{HW}) : \sum_{type=1}^{N_{impl}} usedHW_{type}^{j} \leq 1. \tag{5-13}$$

Resource constraints The HW part of the target architecture is constrained by the quantity of available resources. These resources are separated in four distinct types: Blocks RAM (BRAM), DSP blocks (DSP), Flip-flops (FF), Look Up Tables (LUT). The resource constraints that guarantee the availability of resources are defined as:

$$\begin{aligned} &\forall r \in \{\text{BRAM, DSP, FF, LUT}\} : \\ &\sum_{j=1}^{N_{HW}} \sum_{type=1}^{N_{impl}} usedHW_{type}^{j} \times RscCost_{r}^{type} \leq avlbRsc_{r}, \end{aligned} \tag{5-14}$$

where $RscCost_{r}^{type}$ is the cost in resource r of implementation $type$, and $avlbRsc_{r}$ the available resource of type r in the architecture. These constraints prevent the occurrence of invalid configurations.

3.2-2 Cost functions

Time cost function Two distinct cost functions related to execution time are defined: (5-15) and (5-16) compute execution time of a HW thread and a SW thread, respectively.

$$\begin{aligned} &\forall j \in (1 \dots N_{HW}) : \\ &\qquad T_{HW}^{j} = tiles_{HW}^{j} \times \sum_{type=1}^{N_{impl}} (T_{comp_{HW}}^{type} \times usedHW_{type}^{j}) + j \times T_{spawn} \end{aligned} \tag{5-15}$$

$$\begin{aligned} &\forall i \in (1 \dots N_{SW}) : \\ &\qquad T_{SW}^{i} = tiles_{SW}^{i} \times \sum_{type=1}^{N_{ISA}} (T_{comp_{SW}}^{type} \times usedSW_{type}^{i}) + (N_{HW}^{used} + i) \times T_{spawn} \end{aligned} \tag{5-16}$$

These functions are built upon the computation time parameters, the number of allocated tiles and the time needed for thread spawning. Finally, the total execution time of the application T_t is estimated by:

$$T_t = \max \left[\max_{j \in (1 \dots N_{HW})} (T_{HW}^{j}), \max_{i \in (1 \dots N_{SW})} (T_{SW}^{i}) \right] \tag{5-17}$$

Static Energy By denoting P_{base} the static power of the base architecture (architecture with no accelerator built in the PL fabric and the simplest ISA used), $\Delta SW_{Pstat}^{type_{ISA}}$ the additional static power introduced by the use of ISA $type_{ISA}$, and $\Delta HW_{Pstat}^{type_{Impl}}$ the additional static power introduced by the resources used by HW implementation $type_{Impl}$ in the PL fabric, the static energy is computed by:

$$E_{stat} = T_t \left(P_{base} + \sum_{i=1}^{N_{SW}} \sum_{type_{ISA}=1}^{N_{ISA}} \Delta SW_{Pstat}^{type_{ISA}} \times usedSW_{type_{ISA}}^{i} + \sum_{j=1}^{N_{HW}} \sum_{type_{Impl}=1}^{N_{impl}} \Delta HW_{Pstat}^{type_{Impl}} \times usedHW_{type_{Impl}}^{j} \right) \tag{5-18}$$

This expression includes non-linear terms, which makes the MILP approach impossible. To solve this issue, a variable $T_{HW}^{type,j}$ is introduced. The constraints expressed in (5-19) set $T_{HW}^{type,j}$ equal to T_t

when implementation $type$ is used on HW core j, and to 0 in the other cases. For this purpose, a large integer K is used to set the constraints to 0 when necessary.

$$\begin{aligned} T_t - T_{HW}^{type,j} &\leq K \times (1 - usedHW_{type}^{j}), \\ T_{HW}^{type,j} - T_t &\leq K \times (1 - usedHW_{type}^{j}), \\ T_{HW}^{type,j} &\leq K \times usedHW_{type}^{j}. \end{aligned} \tag{5-19}$$

The same approach is applied for SW ISA with the variable $T_{SW}^{type,i}$ and the following set of inequalities:

$$\begin{aligned} T_t - T_{SW}^{type,i} &\leq K \times (1 - usedSW_{type}^{i}), \\ T_{SW}^{type,i} - T_t &\leq K \times (1 - usedSW_{type}^{i}), \\ T_{SW}^{type,i} &\leq K \times usedSW_{type}^{i}. \end{aligned} \tag{5-20}$$

By including $T_{HW}^{type,j}$ and $T_{SW}^{type,i}$ in (5-18), the static energy can be expressed without non-linear terms as:

$$E_{stat} = T_t \times P_{base} + \sum_{i=1}^{N_{SW}} \sum_{type_{ISA}=1}^{N_{ISA}} (\Delta SW_{Pstat}^{type_{ISA}} \times T_{SW}^{type_{ISA},i}) + \sum_{j=1}^{N_{HW}} \sum_{type_{Impl}=1}^{N_{impl}} (\Delta HW_{Pstat}^{type_{Impl}} \times T_{HW}^{type_{Impl},j}). \tag{5-21}$$

Energy cost function The total energy cost E_t is computed by summing the dynamic energy dissipated by computations and communications, and the static energy dissipated during execution time:

$$\begin{aligned} E_t \;=\; & E_{stat} \\ & + \sum_{i=1}^{N_{SW}} (tiles_{SW}^{i} \times \sum_{type_{ISA}=1}^{N_{ISA}} E_{comp_{SW}}^{type_{ISA}} \times usedSW_{type_{ISA}}^{i} + E_{com_{SW}}) \\ & + \sum_{j=1}^{N_{HW}} (tiles_{HW}^{j} \times \sum_{type_{Impl}=1}^{N_{impl}} E_{comp_{HW}}^{type_{Impl}} \times usedHW_{type_{Impl}}^{j} + E_{com_{HW}}). \end{aligned} \tag{5-22}$$

3.2-3 Objective functions

The solver first builds the configuration vector $\vec{C}$ using the constraints and then selects the best solution using the cost functions according to the optimization objective. Two objectives are defined: minimizing the overall execution energy or minimizing the overall execution time. These objective functions are defined as:

$$\vec{C}_{energy} = \min_{\forall \vec{C}} \left[E_t(\vec{C}) \right] \tag{5-23}$$

$$\vec{C}_{time} = \min_{\forall \vec{C}} \left[T_t(\vec{C}) \right] \tag{5-24}$$

The output of the MILP optimization is the best configuration vector to achieve the objective and an estimation of its characteristics (energy consumption and execution time). Fig. 5-3 and Fig. 5-4 page 85 plot the solution obtained with the MILP-based method (configuration plotted with a red ×) for the matrix multiplication kernel and for the stencil kernel, respectively. It can be noticed that this solution is always cost-optimal and even not covered through brute-force exploration in a limited amount of time.

4 Experimental Setup

This section describes the two application kernels used to validate the proposed tiled-DSE method. The implementation methods involved in the design of HW and SW computation tiles are described and the characteristics of the HW block obtained are detailed. Then, the measures are exposed for one implementation to show the measurement infrastructure and the impact of the HW/SW distribution.

4.1 Application kernels

The first test case is a matrix multiplication (Matmult) application. Input and output matrices are of size 512×512. Matrix computation is subdivided in 256 tiles of size 32×32. Each tile reads full rows or columns in the input matrices and writes tile results in the output one. The second test case is a Stencil computation based on a filter of size 4×4, which is successively applied 10 times. The input and output matrices are of size 542×542 and 512×512, respectively. The application is tiled with overlapping approach to prevent inter-tile dependency, which results in 256 independent tasks to be computed.

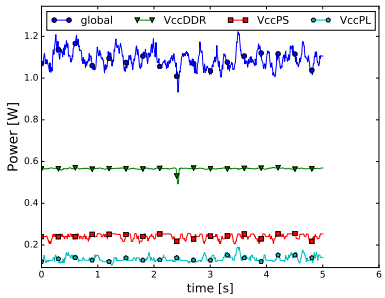
4.2 Measurement infrastructure

The experimental setup used is the same as the one used for the validation of the previous communication power model (cf. 2.3). The measures obtained are used to compute the instantaneous power and the energy variation over the application execution. Fig. 5-5 displays the power curves obtained for test examples of the Matmult application with 9 different HW usages. The curves show the three more representative power rails: VccPS powers the dual cortex A9 processors, VccPL the computational core in the PL fabric, and VccDDR the external DDR memory. The last curve *global* displays the total power consumption over the 7 rails. In these plots, the computation kernel of the Matmult application is executed 5 times on each workload balances between SW and HW resources (from full SW execution up to full HW). These curves show that the timing resolution of the power measurement is small enough to precisely display each computation step involved in the test. In Fig. 5-5d, the HW computation finishes before the SW one for each execution. The SW computation is therefore the bottleneck of this configuration. On the other hand, the HW part is the computation bottleneck in Fig. 5-5g.

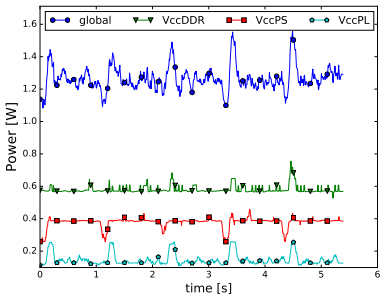
4.3 Hardware implementations

Hardware exploration was performed to obtain several implementations with different characteristics in terms of resources used and computation latency. Only representative implementations were selected among the HW design space. Hardware implementations were generated using Vivado HLS tool *(2015.4 HLX Edition)* [Inc15].

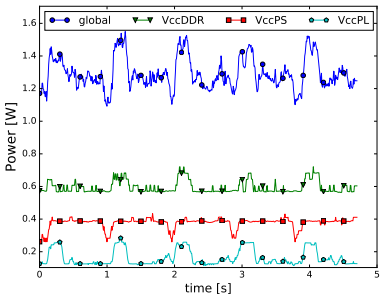
The HW implementations use BRAM scratchpad to maximize the sequential access in memory and make profit of the DMA and burst features. The implementations are specified in a manner to maximize the overlapping between the communications and the computations. Table 5-1 gives the latency and the hardware resources for each implementation of the Matmult and Stencil tiled computations. All these designs are pipelined, but the parallelism degree of the inner loop-nest is different. The implementation name in the table expresses the parallelism degree of the three



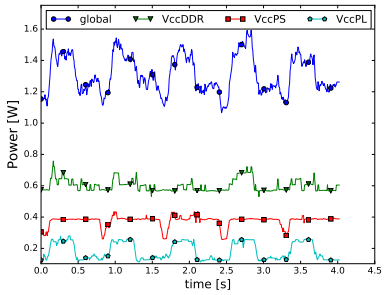
(a) **Matmult Static power consumption**



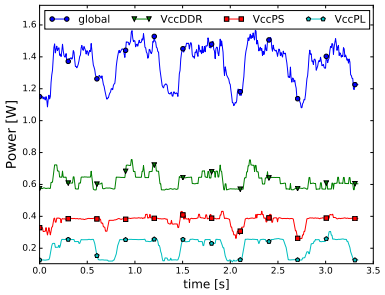
(b) **Matmult** 12.5 % **performed by HW**



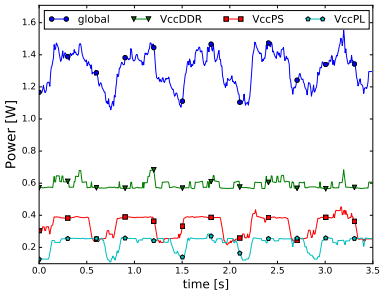
(c) **Matmult** 25 % **performed by HW**



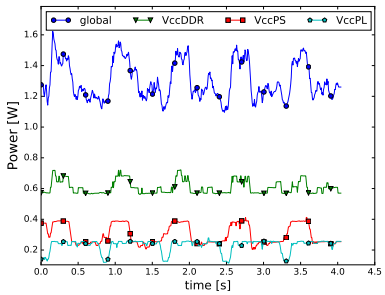
(d) **Matmult** 37.5 % **performed by HW**



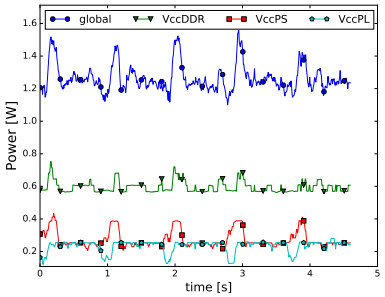
(e) **Matmult** 50 % **performed by HW**



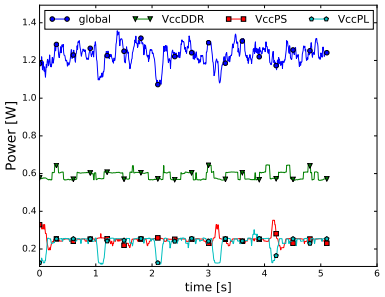
(f) **Matmult** 62.5 % **performed by HW**



(g) **Matmult** 75 % **performed by HW**



(h) **Matmult** 87.5 % **performed by HW**



(i) **Matmult** 100 % **performed by HW**

Figure 5-5 – **Power measurement traces for Impl. LnP** 118 **(cf. Sub-Section 4.3).**

inner loop-nests. For example, the algorithm displayed in Alg. 3 shows matrix multiply structure with labeled loop-nests. In the following the loop-nest iteration space is modified to express more parallelism. Each modification leads to a new HW implementation named "LnP *ABC*", where the number used in *A* position reflects the parallelism expressed in the loop labeled by LnP_Axx and respectively *B* for Lnp_xBx and *C* for Lnp_xxC. LnP stands for *Loop-nest Parallelism*. For instance, LnP 248 means that 2 iterations of the third inner loop (label LnP_Axx) are executed in parallel, 4 for the second one (label LnP_xBx), and 8 for the first one (label LnP_xxC).

The HLS tool used allows the user to add directives (through the use of *#pragma*) to help the tool to find the available parallelism and to generate particular HW structure. In our case, the use of the *LOOP_UNROLL* directive seems natural to express the desired parallelism of the loop-nest. However, the tool is highly conservative on the data dependencies and instructions scheduling. The only way to express with precision the underlying desired HW structure is to rewrite the input C code to be more expressive. For example with the matmult application, we want to break the multiply and accumulate chain (cf. Fig. 5-6a) in a reduction tree (cf. Fig. 5-6b). To generate this reduction tree, we need to manually change the inner loop iteration space and add temporary variables that express

Apps.	Impl.	Lat. [Mcycle]	Resources [%]			
			BRAM	DSP	FF	LUT
Matmult	LnP 114	140	25%	2%	2%	5%
	LnP 118	73	25%	4%	2%	6%
	LnP 128	40	25%	9%	3%	10%
	LnP 148	23	25%	18%	6%	16%
	LnP 248	14	26%	36%	11%	30%
	LnP 448	12	30%	59%	19%	47%
Stencil	LnP 114	12	24%	39%	10%	42%
	LnP 244	7	21%	75%	20%	72%
	LnP 384	6	24%	100%	28%	96%

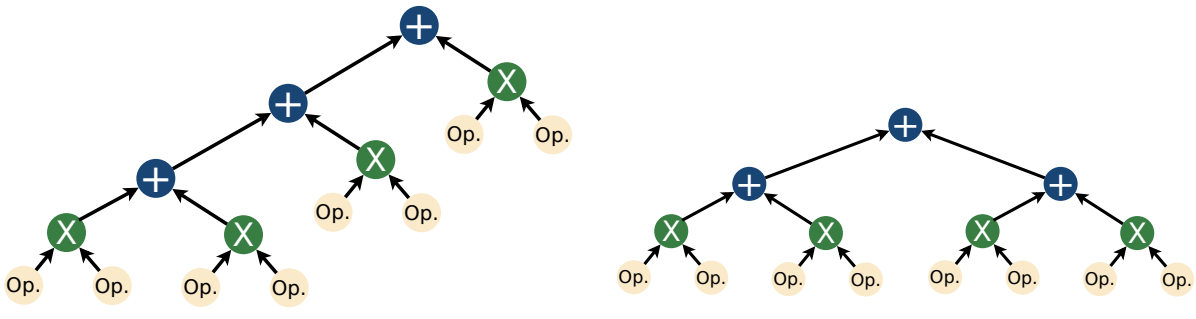
Table 5-1 – **Latency and resource usage of the different HW implementations.**

Data: $Mtx_A[N_t][N]$, $Mtx_B[N][N_t]$, $Mtx_{Out}[N_t][N_t]$
LnP_Axx: **for** *(uint* $i = 0$; $i<N_t$; $i++$*)* **do**
  LnP_xBx: **for** *(uint* $j = 0$; $j<N_t$; $j++$*)* **do**
    float $tmpAcc = 0$;
    LnP_xxC: **for** *(uint* $k = 0$; $k<N$; $k++$*)* **do**
      $tmpAcc = Mtx_A[i][k] \times Mtx_B[k][j]$;
    end
    $Mtx_{Out}[i][j] = tmpAcc$;
  end
end

Algorithm 3: Tiled matmult loop nest

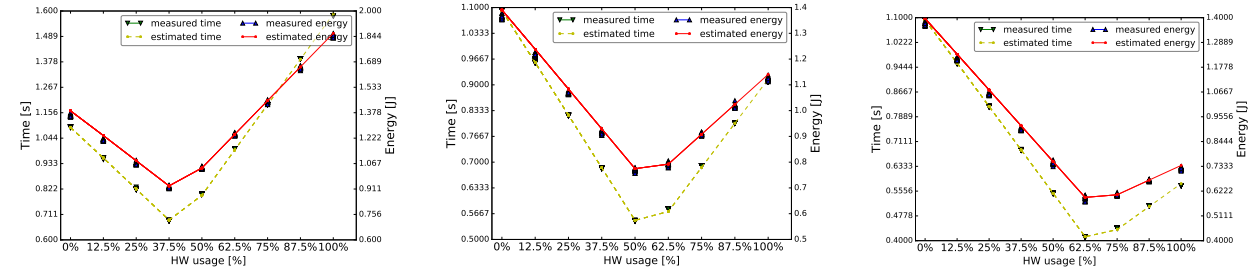
each desired computation independently. Some code snippet is given in Appendix A to illustrate the result of the rewrite process for two implementations.

For a fair comparison between the HW implementations, the loop-nest flattening has been stopped when the use of *DSP* block resources becomes prohibitive without a real incidence on latency. This could be seen on Table 5-1, the exploration was stopped when the ratio *Latency versus used DSP resources* presents an inflection point. In this table, the latency represents the number of cycles needed to compute the 256 tiles of the output matrix. The values are given in million cycles with an HW frequency of 100 MHz.

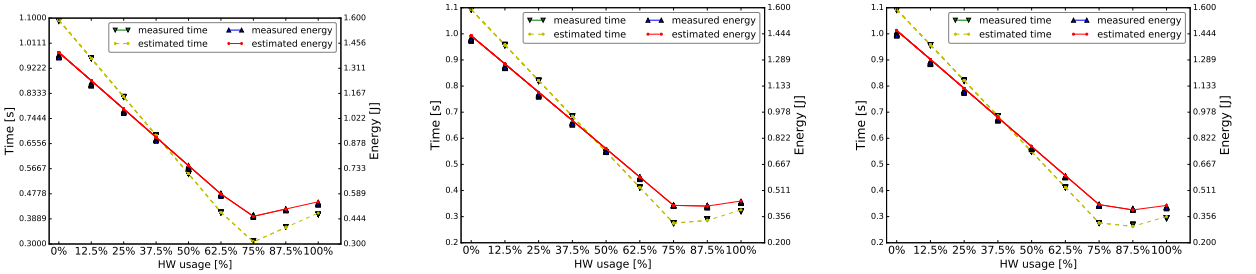


(a) **Basic structure** (b) **Target reduction tree structure**

Figure 5-6 – **Matmult inner loop multiply and accumulate chain.**

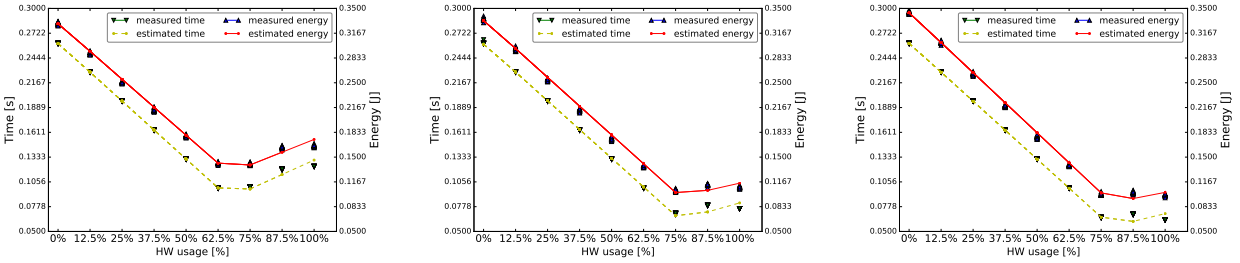


(a) **Impl. LnP** 114 (b) **Impl. LnP** 118 (c) **Impl. LnP** 128



(d) **Impl. LnP** 148 (e) **Impl. LnP** 248 (f) **Impl. LnP** 448

Figure 5-7 – **Energy consumption and execution time versus HW usage: measured and estimated values for Matmult.**



(a) **Impl. LnP** 114 (b) **Impl. LnP** 244 (c) **Impl. LnP** 384

Figure 5-8 – **Energy consumption and execution time versus HW usage: measured and estimated values for Stencil.**

4.4 Software implementations

The software implementation is written in order to expose instruction parallelism and to maximize the cache reuse. Compiled with gcc and the *-Ofast* flag, the obtained binary code uses the dedicated NEON instructions to draw the full power of the ARM architecture. The obtained SW takes approximately 8.5×10^{-3} s to compute one tile for the mamult application and 2×10^{-3} s for the stencil application on one Arm core running at 800 MHz.

5 Exploration Results

In this section, we present the parameter extraction results and we show their accuracy. In addition the MILP optimization results are displayed for both applications, and the obtained configurations are launched on real architecture. The measured execution values are then compared to the estimated one to obtain the accuracy of the given tiled-DSE method.

5.1 Parameter extraction

The set of sample configurations was launched 20 times for each HW implementation. The execution power measurement traces were then parsed and processed to obtain a set of test points composed of the sample configuration, the average energy consumption and the execution time. First, an estimation of the communication cost is computed for each sample configuration. Then, the cost functions described in (5-4) and (5-6) are used with an LSR algorithm to extract time cost parameters and energy cost parameters. These functions will be used to explore the design space.

Measured values are compared to the estimation results computed using the previous extracted parameters. The extraction and estimation were performed for each HW implementation of the Matmult and Stencil applications. The measurements are fully automatized and take $336\ s$ for the Matmult and $122\ s$ for the Stencil, which is adequate since parameter extraction is executed only once. The results are shown in Fig. 5-7 for Matmult application and in Fig. 5-8 for Stencil application. These curves display, for each implementation, the measured and estimated values for energy consumption and execution time for different HW/SW tile distributions. These results show that the error between measured and estimated values is low, showing the good accuracy of the extracted parameters. SW cost parameters (*i.e.* HW usage = 0%) are independent of the associated HW implementation. Moreover, another accuracy indicator of the proposed extraction method is that the error on SW parameters has a standard deviation of 7.2×10^{-6} for Matmult application and of 1.2×10^{-6} for Stencil. In addition, these curves show that the HW/SW tile distribution that minimizes the energy consumption clearly depends on the used implementation.

Conf.	Thread ID					
	SW0	SW1	HW0	HW1	HW2	HW3
Implem.	SW	SW	LnP 248	LnP 248	LnP 148	none
N_{tiles}	9	9	87	85	66	0

Table 5-2 – **Matmult configuration after MILP optimization (energy objective).**

5.2 MILP optimization

In this section, the application parameters and the previously extracted cost parameters feed the MILP models. Then, the optimization problem is solved based on the energy cost function (5-24) using *Gurobi* MILP solver [Opt16] on an *Intel i7 Haswell-ult* processor running at 2.10 GHz. The MILP solution is obtained after 18980 simplex iterations in $\simeq 0.73s$ for the Matmult application and after 348 simplex iterations in $\simeq 0.03s$ for the Stencil application. This difference is mainly due to the higher number of HW implementations available for Matmult application, which enlarges the design space size. However, even for quite complex kernels and such large design space, these results demonstrate the scalability of the MILP approach. Table 5-2 shows the MILP result configuration for the Matmult application with the tile distribution across the cores and the implementations used, obtained after MILP optimization. 18 tiles are equally distributed on the SW core, the remaining tiles

Conf.	Thread ID					
	SW0	SW1	HW0	HW1	HW2	HW3
Implem.	SW	SW	LnP 114	LnP 114	none	none
N_{tiles}	25	24	104	103	0	0

Table 5-3 – **Stencil configuration after MILP optimization (energy objective).**

Conf.	Thread ID					
	SW0	SW1	HW0	HW1	HW2	HW3
Implem.	SW	SW	LnP 248	LnP 248	LnP 048	none
N_{tiles}	11	11	85	84	65	0

Table 5-4 – **Matmult configuration after MILP optimization (time objective).**

Conf.	Thread ID					
	SW0	SW1	HW0	HW1	HW2	HW3
Implem.	SW	SW	LnP 114	LnP 114	none	none
N_{tiles}	25	25	103	103	0	0

Table 5-5 – **Stencil configuration after MILP optimization (time objective).**

are distributed on 3 HW cores. Implementation *LnP 248* is used on 2 HW cores, the third one uses a smaller implementation to fulfil the remaining space in the FPGA PL fabric. In this case, the number of HW cores used is limited by the available FPGA resources and not by the upper-bound due to the memory bandwidth. Table 5-3 gives the MILP result configuration for Stencil application. As the stencil implementations use more resources than the Matmult ones, only 2 HW cores are used. Table 5-4 and Table 5-5 show the result of the MILP optimization configured with the time objective function. As we can see, the obtained results are quiet similar to the energy objective function. This phenomenon could be easily explained by the fact that we considered the static power consumption of the DDR memory in our experiments, which leads to a large global static power consumption that reduces the incidence of dynamic power consumption. Therefore, minimizing the execution time is a good approach to reduce the energy consumption. In the following, we will focus on the results obtained with the energy objective function.

5.3 Precision and gain factors

In this section, the performance of the MILP configuration is compared with the best sample configuration among the implementations and with two basic configurations: Full SW and Full HW. Table 5-6 shows the resources used by each configuration, the estimated execution time and estimated energy consumption for both Matmult and Stencil applications. Since the MILP configuration can use more than one HW core, the total amount of resources used could be greater than with the sample configuration (cf. Table 5-1). This is the case with the Matmult application. On the other hand, the MILP configuration for Stencil application uses two smaller implementations than the one used in the sample configuration, which results in more BRAM used but less DSP blocks used. The

	Conf.	Resources [%]				Time [s]	acc. gain	Energy [J]	energy red.
		BRAM	DSP	FF	LUT				
Matmult	Full SW	0%	0%	0%	0%	1.21	−307.7%	1.59	−199.4%
	Full HW	30%	59%	19%	47%	4.1×10^{-1}	−5.1%	5.6×10^{-1}	−3.7%
	Sample Conf.	30%	59%	19%	47%	3.9×10^{-1}	0%	5.4×10^{-1}	0%
	MILP	76%	90%	28%	76%	2.290×10^{-1}	**41.3%**	3.558×10^{-1}	**34.1%**
Stencil	Full SW	0%	0%	0%	0%	3.2×10^{-1}	−150%	4.2×10^{-1}	−135.9%
	Full HW	24%	100%	28%	96%	1.28×10^{-1}	0%	1.78×10^{-1}	0%
	Sample Conf.	24%	100%	28%	96%	1.28×10^{-1}	0%	1.78×10^{-1}	0%
	MILP	48%	78%	20%	84%	1.121×10^{-1}	**12.4%**	1.566×10^{-1}	**12%**

Table 5-6 – **Exploration results.**

Application		Time [s]	err. [%]	Energy [J]	err. [%]
Mat.	Est.	2.29×10^{-1}	3.35%	3.558×10^{-1}	5.03%
	Meas.	2.369×10^{-1}		3.746×10^{-1}	
Sten.	Est.	1.121×10^{-1}	9.58%	1.566×10^{-1}	10.25%
	Meas.	1.24×10^{-1}		1.745×10^{-1}	

Table 5-7 – **Comparison between estimation and measure.**

acceleration gain and energy reduction are the ratio between a given configuration and the most energy efficient sample configuration taken as reference. The impact of static energy on the total energy is not negligible. Therefore, energy reduction is linked with acceleration factor. It is noteworthy that the MILP optimization ensures an energy saving of approximately $34\,\%$ for the Matmult application and approximately $12\ \%$ for the Stencil application.

To validate the proposed approach, time and energy of the MILP configuration are compared with the real measurements of the same configuration over the Zynq board. The measured values are obtained over 40 independent executions of the MILP configuration. Table 5-7 shows both estimated and measured values for the two applications. The average error for Matmult application is around $5\,\%$, which is quite low for power measurement on real architecture. For Stencil application, the average error is higher, which is due to the execution time difference. Indeed, as the matrix multiplication execution time is longer than for the Stencil, power measurement inaccuracy is less mitigated in the case of the Stencil application.

Table 5-8 summarizes the results of the MILP-based DSE exploration method for tiled applications, presented in this chapter.

6 Conclusion

This chapter introduced a new exploration method for building energy-efficient accelerators on heterogeneous architecture. The method targets tiled computation kernel and is based on the measurements of a tiny subset of the design space. These measurements are then injected into two extraction functions to obtain analytical formulations of the execution time and energy consumption of the computation kernel. Thereafter, this information is injected in the fast power model detailed in Chapter 4. Then, the tile distribution constraints are captured using an MILP formulation and a solver is used to obtain the best solution. This methodology was tested on two application kernels, a matrix multiplication and a stencil computation, on a Zynq-based heterogeneous architecture. These experiments show that the acceleration and energy saving obtained are between $12\ \%$ to $41\ \%$ compared to the best sample configuration. Furthermore, the estimation accuracy is within $5\ \%$ to $10\ \%$, which is quite acceptable for real hardware measurements. These results open up new opportunities for future CAD tools. Especially, this MILP approach could be extended with a multi-step exploration to accelerate global applications composed of multiple computation kernels and to obtain an energy-efficient mapping of a full application on heterogeneous multiprocessor architectures.

	Conf.	Resources [%]				Time [s]			acc. gain	Energy [J]			energy red.
		BRAM	DSP	FF	LUT								
Matmult	Full SW	0%	0%	0%	0%	1.21			4.077	1.59			2.944
	Full HW	30%	59%	19%	47%	4.1×10^{-1}			1.051	5.6×10^{-1}			1.037
	Sample Conf.	30%	59%	19%	47%	3.9×10^{-1}			1.0	5.4×10^{-1}			1.0
	MILP	76%	90%	28%	76%	est.	2.290×10^{-1}	err. 3.35%	0.587	est.	3.558×10^{-1}	err. 5.03%	0.659
						meas.	2.369×10^{-1}			meas.	3.746×10^{-1}		
Stencil	Full SW	0%	0%	0%	0%	3.2×10^{-1}			2.5	4.2×10^{-1}			2.359
	Full HW	24%	100%	28%	96%	1.128×10^{-1}			1.0	1.78×10^{-1}			1.0
	Sample Conf.	24%	100%	28%	96%	1.128×10^{-1}			1.0	1.78×10^{-1}			1.0
	MILP	48%	78%	20%	84%	est.	1.121×10^{-1}	err. 9.58%	0.876	est.	1.566×10^{-1}	err. 10.25%	0.880
						meas.	1.24×10^{-1}			meas.	1.745×10^{-1}		

Table 5-8 – **Global overview of the energy-driven accelerator exploration results.**

Chapter **SIX**

HMPSOC EMULATION PLATFORM

Contents

1 Motivations

In the previous section we have presented the work done toward the adoption of an energy-aware design flow over HMpSoC. The experiments involved in this work have highlighted some disadvantages that prevent their large-scale utilization on numerous architecture and applications. In fact, in Chapter 4, the involved experiments have shown that the validation of power model on a broad range of architectures could be time consuming, especially because it requires some development on the target architecture. The development of a test infrastructure on a complex HMpSoC running an OS requires a set of prerequisites that is not possible on every architecture. For example, the experiments done on the MPPA highlight the necessity to get a precise power measure system, precision that could not be met on our evaluation board. On top of that, when the target architecture includes the right power measurement sensor, additional develoments need to be done to enable the measurement automation alongside the tested application. Furthermore, when these requirements are met, such as in Chapter 5 where we reuse the previous measurement infrastructure built for the Zynq architecture, we have met another difficulty related to the tested application. The available benchmark sets (cf. [Bie+08; ILG10; PHB13; Har+08; Gus+10]) provide a large bunch of applications. However, each benchmark set has its own target application structure and porting them on the dedicated architecture and execution framework is not straightforward and requires a consistent amount of work, even for small applications such as the matmult and stencil computation kernels that we use in this thesis.

These drawbacks prevent the test of new energy-aware algorithms, such as the approach proposed in Chapter 5, on numerous applications and architectures. However, the work already done in this thesis around the fast power modeling could be gathered in an emulation platform that meets

the test requirements of algorithm mapping on HMpSoC. For this purpose, we decided to built a configurable HMpSoC emulation platform that enables the test of energy-aware mapping algorithms in a more automated fashion. The proposed emulation platform and framework should meet the following properties:

- The execution time on the emulation platform should be low to enable the emulation of complex HMpSoC in a reasonable time.
- The emulation platform should embed a fine grain configuration layer to describe a large panel of HMpSoC architectures.
- Changing the configuration should be performed as fast as possible.
- The emulation platform should embed power modelling facilities to report the energy consumption of the executed application.
- The emulation platform should be shipped with an execution infrastructure that enables the spreading of job/task on the clusters.
- The emulation platform should be associated with a bunch of representative applications.
- The provided application execution mapping should be customized through directives.
- The provided application and mapping information should be executable on the emulation platform without extra manipulation from users.

The long term objective behind this emulation platform and the application bundle is to integrate them in the back-end of an existing task-mapping framework. This will provide to the task-mapping algorithm design field a full and easily usable test framework and then ease the large-scale test of these algorithms.

This chapter is organized as follows. In Section 2, we present the available emulation technologies and we describe the emulation structure built alongside with the configuration facilities. Then, we depict the cluster management methodology used and the execution framework structure. Section 3 introduces a set of representative computation kernels – called dwarf – that can be used to build an infinite set of representative applications. The kernel implementation and the association method are described in details. The power modelling facilities embedded in the emulation platform are then tested with various generated applications. Finally, we discuss the emulation platform potentials in Section 4.

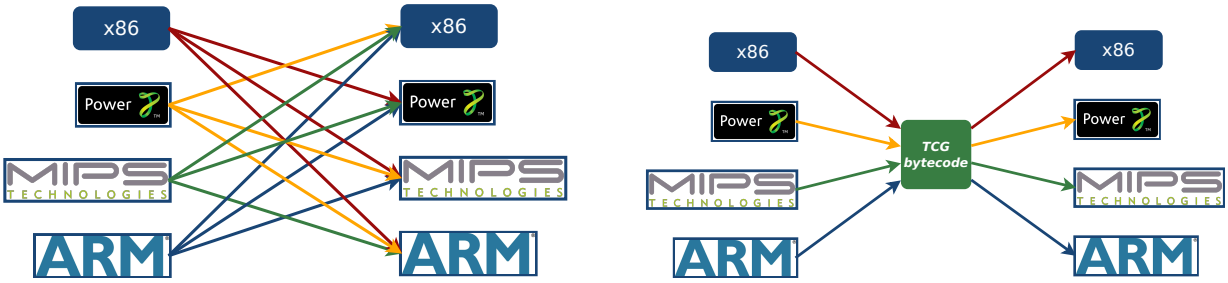
2 Architecture emulation layer

In this section, we present some available emulation technologies and their advantages and drawbacks to motivate our choice. Then, we introduce the structure of the emulation platform designed and expose the difficulties encountered.

2.1 Underlying technologies

2.1-1 QEMU

Quick EMUlator (QEMU) [Bel05] is a generic and open-source machine emulator and virtualizer created in 2003 by Fabrice Bellard. QEMU uses a dynamic binary instruction translator, which al-



(a) **One stage translation mechanism**

(b) **Two stage translation mechanism used in TCG**

Figure 6-1 – **Advantage in number of translators implied by the two-stage mechanism used in TCG.**

lows the execution of a code compiled for one target ISA on another host ISA. The ISAs supported by QEMU are wide and include x86, SPARC, ARM, MIPS, Power PC, Microblaze. QEMU could be used in two operation modes: full emulation mode or user mode. In the full emulation mode, QEMU emulates one or more processors and hardware peripherals. In the user mode, only a process compiled for one ISA is emulated. Since 2003, a lot of people contributed to QEMU and the source codes are often updated [BC].

Tiny code generator QEMU uses a Just In Time (JIT) dynamic recompilation of the target code for the host architecture. This JIT translator, called Tiny Code Generator (TCG), is separated in a front-end that translates the target ISA in intermediate bytecode, and in a back-end that translates the bytecode instructions to the target host instructions. With this approach, instead of needing a specific point-to-point instruction translator for each target to host ISA pair, one translator to intermediate bytecode is needed for each target ISA, and one translator from bytecode is needed for each host ISA. This two-stage mechanism greatly reduces the number of translators that have to be implemented. This feature is highlighted on Fig. 6-1.

On top of that, TCG uses a block translation mechanism, which translates group of several instructions at a time called translation block. The translation block splitting relies on the same properties as basic block used in standard compiler. They are composed of a list of subsequent instructions containing a branch instruction only at its end. Each translation block is then read, disassembled, and translated into bytecode blocks. At this stage, some compiler optimizations are run to removed the useless instructions and dead variables. Then the bytecode blocks are translated into native instruction blocks for the host architecture in order to be executed. The recently translated blocks are stored in a kind of translation cache to be easily reuse without rerunning the translation stages again. This mechanism enhances the emulation execution time and enables QEMU to achieve good emulation performance.

This modular structure allows QEMU to emulate from and to a broad range of ISA. QEMU comes with many peripherals and can build a large panel of architectures. With the use of the TCG and the translation block cache, QEMU can easily boot a full-flavor operating system such as linux within a reasonable time. These elements make QEMU a good processor ISA emulation solution for our needs. Furthermore, the high development activity around QEMU provides support and augurs well for a long-term maintenance of the project.

We now need to find a fast HW emulation language that can be coupled with QEMU and can quickly model the behavior of the heterogeneous part of the targeted HMpSoC structure. The SystemC language is a good candidate.

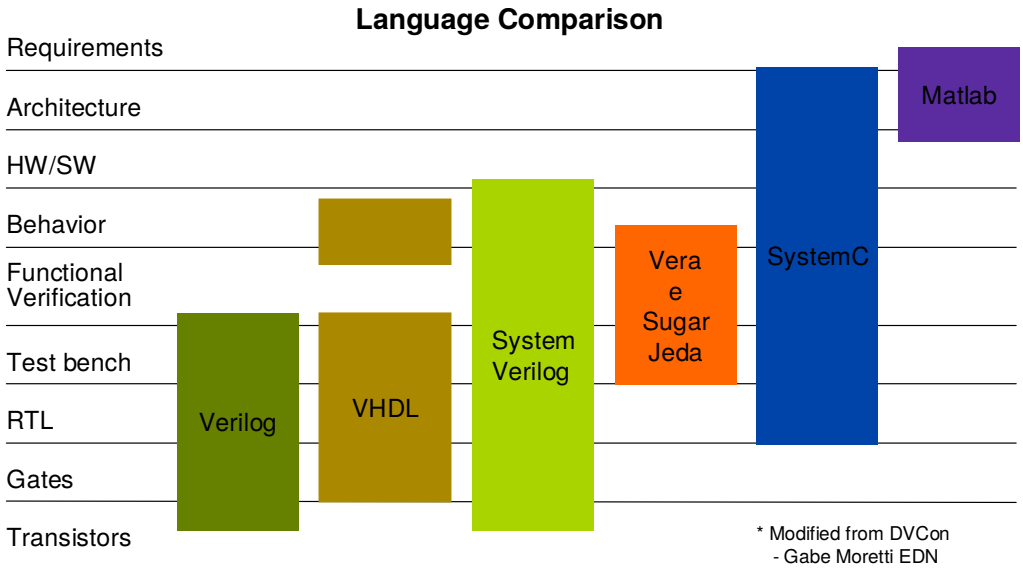


Figure 6-2 – **SystemC representation capabilities compared to other HDLs [Bla+09].**

2.1-2 SystemC

SystemC [Bla+09] is an open source-class library built upon C++. SystemC is a common language used to model both HW systems and SW applications. It has the advantage of integrating HW and SW modules at different abstraction levels, such as register level or transaction level. This allows for SystemC models to perform HW/SW co-simulation of complex systems with high performance.

Figure 6-2 depicts the representation scope of SystemC compared to other Hardware Description Languages (HDLs) and high-level modelling languages. As we can see, SystemC can be used to model a wide range of abstraction levels, from simple system requirements down to detailed RTL description levels.

The SystemC standard is developed in a collaborative and open environment supported by the Accellera Systems Initiative (formerly called Open SystemC Initiative (OSCI)). This consortium provides a proof of concept simulation kernel for SystemC. This kernel is conformed to the SystemC standard and is implemented in a mono-threaded fashion. Others simulation kernels with multi-threaded implementations are proposed by Electronic Design Automation (EDA) vendors, but they require commercial licenses and so they cannot be used in the context of this work.

Regardless of the implementation, the SystemC simulation kernel is built around two major phases of operation: *elaboration* and *execution*. A third minor phase occurs at the end of simulation and is called *cleanup*. During the elaboration phase, the data structures are initialized, and then the system components contained in the architecture description are registered and linked together. The execution phase, handled by the simulation kernel, coordinates the execution of the different processes to create an illusion of concurrency between the simulated processes. The simulation kernel relies on a list of sensitive events and cooperative scheduling. This means that the simulation kernel can not force a process to stop. Instead, the process runs and then returns the control to the SystemC simulation kernel when it reaches a specific breakpoint.

After the elaboration phase, all the simulated processes are invoked once in a random order. Then, the processes are invoked when an event to which their are sensitive occurs. Multiple processes may begin at the same instant of the simulator time. This is the reason why all the simulation processes are evaluated and then their outputs are updated. The evaluation followed by the value

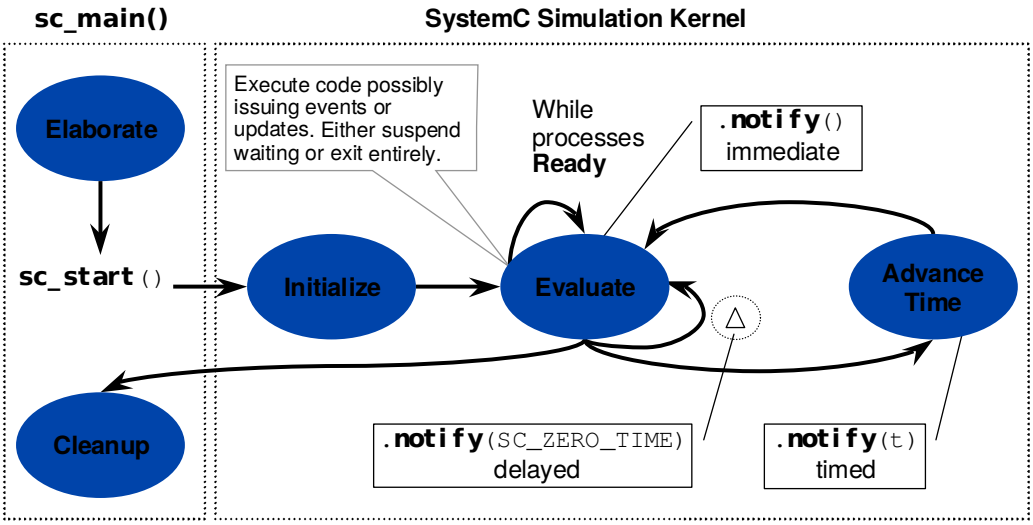


Figure 6-3 – **Overview of the SystemC simulation kernel [Bla+09].**

update is referred to as a *delta-cycle*. When no more processes need to be evaluated at the simulation time, then the simulation time is incremented (*advanced time*). Figure 6-3 depicts the interaction of the operation phases within SystemC simulation kernel.

On top of that, SystemC implements a module hierarchy to deal with very large designs. A module may contain processes and instances of other modules. Modules communicate together through ports that could receive or send data of various types.

In 2009, the OSCI introduces the TLM-2.0 standard which is the second version of the Transaction-Level Modeling standard [Joh09]. The interfaces introduced by TLM-2.0 enhance SystemC simulation performance. In fact, this new communication layer increases the communication level from pin-accurate level to function-call level. With this new communication-centric module interfaces, SystemC is now able to simulate application execution within a described architecture.

The TLM standard consists of a set of core interfaces composed of a global quantum (a time interval used for synchronization), a generic payload, an initiator and a target communication socket. TLM-2.0 classes are layered on top of the SystemC class library. The TLM socket mechanism is shown in Fig. 6-4. The generic payload is composed of the destination address of communication, the data length, the command (read or write), the status, and a payload buffer or a data pointer for Direct Memory Interface (DMI). The initiator socket sends this payload to the target socket which is in charge of the implementation of the callback function managing the generic payload. This new communication mechanism can be implemented with two main time representations: Loosely Time and Approximated Time. The Loosely-Time approach implements a one-point synchronization mechanism and the Approximated-Time approach uses a hand-shake mechanism to tighten the synchronization between modules. Compared to the previous pin-accurate communication, the TLM-2.0 primitive relaxes the inter-module synchronization constraints. To keep each module in the same time windows, a new mechanism call global quantum is introduced with a dedicated manager called quantum keeper. These mechanisms guarantee that each modules is not far from the other, in term of simulation time, more than one quantum. The value of the quantum can be configured during the elaboration phase. TLM-2.0 enhances the simulation speed by $100\times$ up to $100,000\times$ compared to the pin-accurate scheme. Furthermore, this mechanism enhances the module isolation and facilitates the interaction between modules described at different abstraction levels

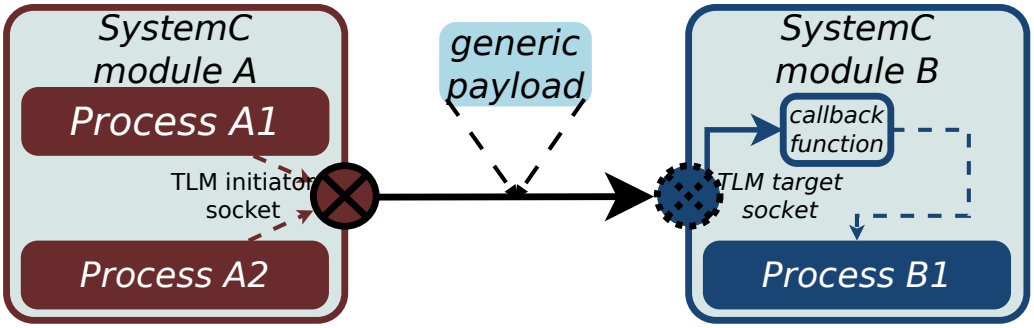


Figure 6-4 – **Overview of SystemC TLM-2.0 transaction mechanisms.**

2.1-3 SystemC and QEMU association

In the past, SystemC emulation of heterogeneous SoC was embedding custom Instruction Set Simulators (ISS) for the SW part of the architecture. In 2009, Marius Monton and greenSocs proposed the first association of SystemC and QEMU, called QEMU-SystemC, to draw the benefit of the QEMU JIT compiler within the context of heterogeneous SoC emulation. Other projects then emerged with the same objective: linking these two technologies. We start by describing the involved synchronization mechanisms. Then, we briefly present the most representative projects.

Synchronization mechanism The time evolution in QEMU and SystemC are represented differently and need to be synchronized when these technologies are associated. In QEMU, time progresses according to the number of instructions executed by the guest processor. As opposite, in SystemC, the time progresses according to the SystemC event occurrences and the described HW platform properties. The synchronization of the two simulators is guaranteed by a strategy of freeze-and-update and achieves good performance. This synchronization mechanism is depicted on Fig. 6-5. On this example, the SystemC simulation kernel is the master. The time in QEMU progresses first for a given number of instructions, then QEMU is frozen. After that, the SystemC simulation runs until the SystemC equivalent time catches up with the QEMU frozen time. Synchronization may also be triggered by SystemC in several cases:

- when the CPU makes I/O accesses into the main emulator,
- when the CPU gets interrupted,
- after the execution of a translation block.

QEMU-SystemC: QEMU-SystemC [MCB09] allows plugging SystemC models into the QEMU emulated platform. This is achieved by introducing a virtual device within QEMU that runs the OSCI SystemC simulation kernel and makes the bridge between QEMU and SystemC through TLM-2.0 socket. This bridge accomplishes the task of synchronization beetween the two simulators, using a strategy of freeze-and-update. With this approach, the QEMU emulation kernel is the master, and the SystemC simulation kernel is a slave. As the result, this connection method is useful when few HW components have to be added to the existing QEMU platforms.

TLMu: Transaction Level eMulator (TLMu) [IC] was released by Edgard E. Iglesias. It is an open-source wrapper for QEMU that integrates into SystemC TLM-2.0 models. With this approach, SystemC simulation kernel remains the master and QEMU fulfils the SystemC API requirements to behave as a standard SystemC modules. The TLMu wrapper loads QEMU as a shared object and

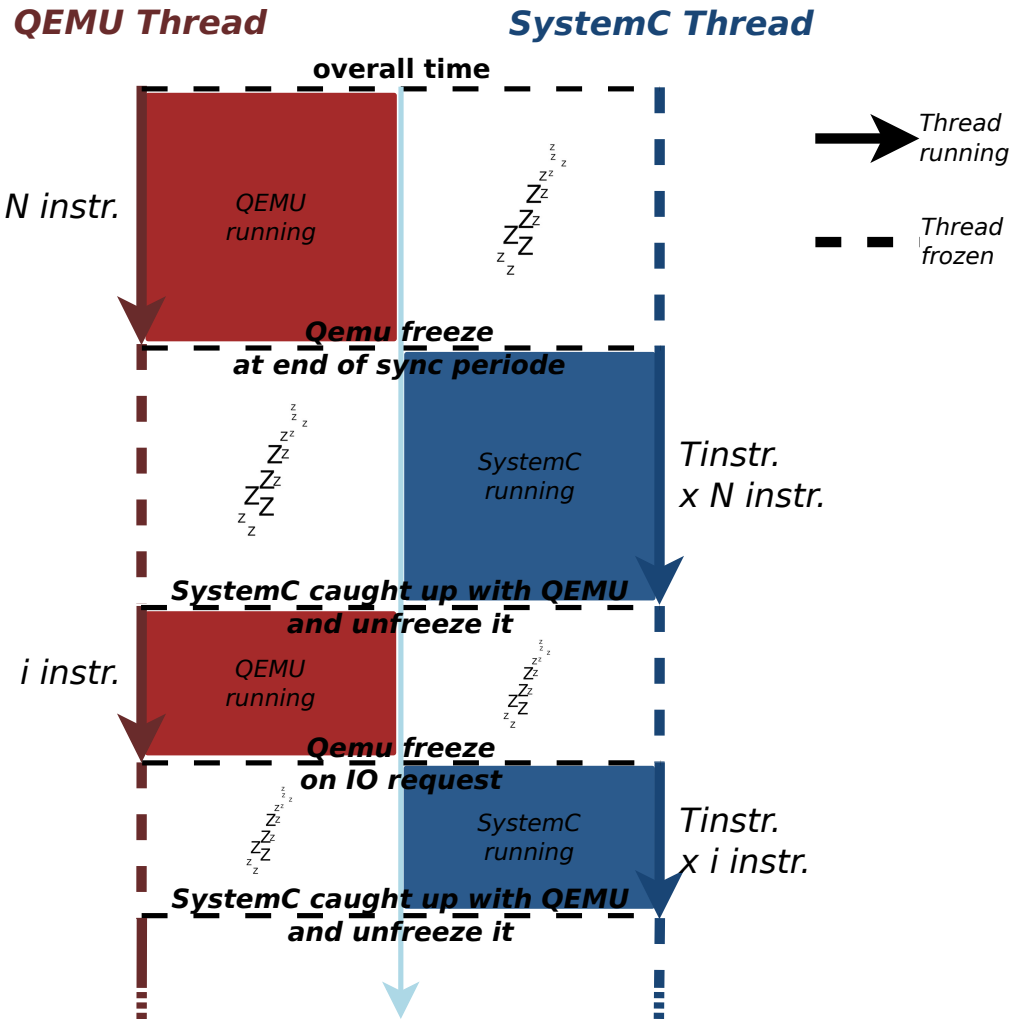


Figure 6-5 – **Overview of the freeze-and-update synchronization strategy.**

defines callback functions. In QEMU, the SystemC environment is registered as a memory region, which, when accessed, uses a callback function to reach the TLM memory bus model to read or write memory spaces in SystemC. This project has received many contributions between 2009 and 2012, the last activity occur in 2014 and then the project seems not to be maintained to match with the evolution of the QEMU project.

QBox: QEMU in a box (QBox) [Del+16] was proposed by greenSocs in 2016. Contrary to the QEMU-SystemC solution, QBox treats QEMU as a standard SystemC module. In the same way as TLMu, the SystemC simulation kernel is the master and the QEMU emulator acts as a slave. QBox is delivered with a wrapper TLM2C and is designed to be associated with other solutions provided by greenSocs such as greenLib. The main objective of greenLib and, in a more general manner of greenSocs, is to provide an overlay upon the SystemC standard with pre-developed models to accelerate the development process of virtual platforms. All the proposed components are designed with fast simulation objectives. At a first glance, these solutions seem really adapted to our aims. Unfortunately, the greenSocs SystemC overlay uses TLM-2.0 features, such as DMI, that prevent the precise monitoring of the communications across the module. And thereby greenSocs solution does not match our needs.

libsystemctlm-soc: This project released by Xilinx [XI] uses the same association mechanism as TLMu and main contributor of this project was previously involved in TLMu (Edgard E. Iglesias). On top of that, this approach isolates QEMU and SystemC simulation kernels in two distinct OS processes and uses a dedicated and well defined API, called libRemote, to manage the communications. This project is under development and currently maintained. It is used by Xilinx to build virtual platforms for their SoC from the Zynq families.

In this section we presented four available solutions that associate and synchronize QEMU and SystemC. At first, QBox and greenSocs infrastructure seem promising for our project. However, the intensive use of DMI prevents the communication monitoring implementation. So, we decided to start our project with the *TLMu* approach, and then to switch to the new *libsystemctlm-soc* released by Xilinx to extend the maintainability of our solution. Furthermore, the Xilinx solution uses a more recent version of QEMU and therefore provides more recent CPU architectures.

2.2 Emulator structure

This section describes the underlying structure of the designed HMpSoC emulation platform. This structure enables to represent a large panel of HMpSoC belonging to the distributed families. It integrates feature to enable multi-threading emulation and so to draw the maximum performance of the host architecture. On top of that, a configuration layer is introduced. This layer allows for the user to easily change the target architecture, without having to compile again the emulator, nor having a strong knowledge of the emulator implementation. The aim of this configuration facilities is to address the broadest range of users as possible. Furthermore, the emulator integrates communication monitoring facilities coupled with a built-in energy estimation based on the communication-based power model.

2.2-1 Cluster structure

The cluster macro structure is composed of a SW part, connected with the HW part through a global communication channel dedicated to configuration and synchronization. Interruption mechanisms are also available between SW and HW. The data communications between the SW and HW parts occur through a shared memory bank. The SW part can be configured with each available QEMU architecture. So it can run multiple ISAs and SMPs with various number of cores. On the other side, the HW part can be loaded with multiple HW accelerators and the size of the shared memory bank is also configurable. The cluster also implements a NoC interface, which receives commands through TLM sockets and then perform the required memory movements in the shared memory bank with DMA. Figure 6-6 depicts a schematic view of the cluster structure. TLM communication channels follow a point-to-point master/slave scheme. On the figure, the master side is depicted with “skM”, and “skS” for the slave side.

On top of that, a communication energy monitoring system is implemented. It is composed of one monitoring system master (msM) and multiple monitoring system slaves (mss). The master module can configure the parameters of each slave to change their energy or time consumption. Each mss is responsible for the monitoring of the cost (time and energy consumption) of each communication occurring on its master TLM socket. For instance, on Fig. 6-6, the mss embedded in HW IP B is responsible of monitoring communications occurring between the IP B and the general purpose communication (GPCom) component. Also, the mss embedded in the memory bank is in charge of monitoring communications of each memory channel with the HW IP, the channel with

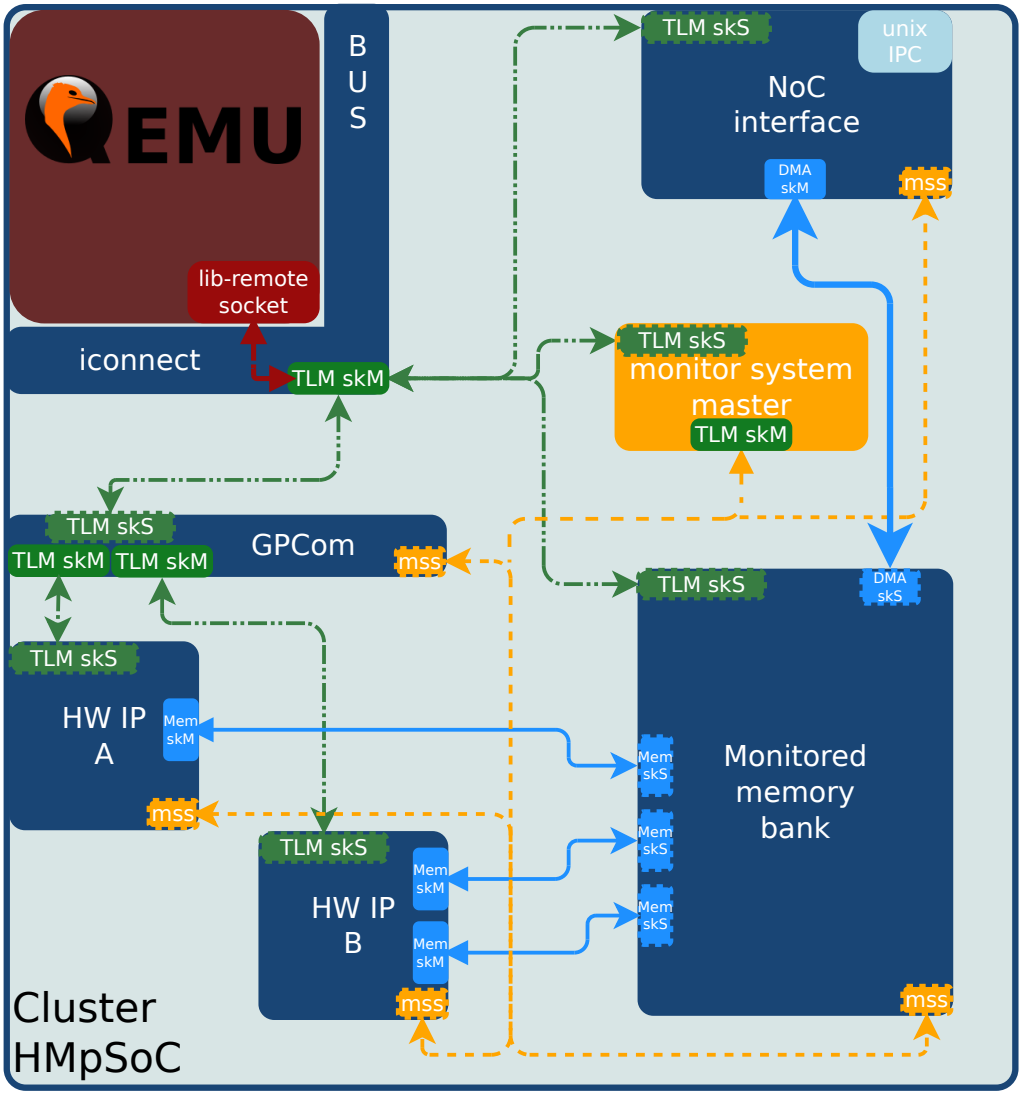


Figure 6-6 – **Overall cluster structure.**

the inter-connect (iconnect) component, and the DMA channel connected to the NoC interface. Each sub-channel has its own set of parameters.

The msM aggregates the consumption counter of each mss and sends this value to SW on demand. With this monitoring facilities, it is easy to configure all communication channel parameters and to implement the communication-based power model introduced in Chapter 4.

2.2-2 NoC structure

The NoC exposes to the cluster some mechanisms that enable communication with a Message Passing Interface (MPI) concepts. Communication packets crossing the NoC contain the target ID, the target cluster address as well as the sender id and address. Various commands are available such as memory request, job request, and synchronization. All packets going through the NoC triggered a tracing mechanism with acknowledgment feature.

The implementation of this NoC as a pure SystemC component would have been easy. Unfortunately, this approach would have gathered all the simulation complexity in the same SystemC simulation kernel. The used SystemC simulation kernel provided by Accelera relies on a mono-threaded application, so, in order to multi-thread the emulation and draw the full power of the host machine,

we need to isolate the simulation of each cluster in a distinct linux process. To achieve this objective, the NoC implementation relies on UNIX Inter-Process Communication (IPC) facilities.

This required cluster isolation in distinct processes raises a synchronization issue. Keeping the time reference of each SystemC simulation kernel synchronized is mandatory. To address this issue, a synchronization mechanism was implemented on top of the IPC. This mechanism relies on a two-phase mechanism: run phase and callback phase. During the run phase, an event is firstly scheduled in each SystemC simulation kernel, with the $NoC\_latency$ latency value. Before its occurrence, the NoC interface of each cluster sends commands, through IPC, to the NoC router. When the previous scheduled event occurs, the NoC interface of each cluster sends a synchronization header to the router and the SystemC simulation freezes. On its side, the router loops on the connected cluster IPC channel. It reads the received command header, finds the target cluster and routes the command. When the router has received a synchronization header from each cluster, it notifies all clusters that the callback phase starts. During this phase, the clusters read their received command and process them. If the processed command asks for data, the cluster generates a callback message that encapsulates the requested data. When all the received commands are processed, the cluster notifies the router with another synchronization header. In parallel, the clusters schedule updates in the local memory for each received callback. The router routes the received callback data messages until it receives all the synchronization callback headers. After that, the router notifies all clusters to start the next cycle and the SystemC simulation kernel of each cluster continues.

This synchronization mechanism introduces a communication time bias of $NoC\_latency$, which is hidden by the real NoC latency of the emulated network. Each command sent on the NoC contains a delta-time that corresponds to the elapsed time since the last synchronization. This delta-time is then used when the commands are processed to delay their effects or pass back to the callback phase to schedule the memory update at the right time. There is also a dynamic communication cost that is computed following the NoC parameters and the distance of the two communicating clusters. Its effects are applied in the same ways as the delta-time. The distance is computed as the number of hopes needed to reach the target cluster from the source one and depends on the NoC topology. The NoC could be configured with the following topology: Ring, Mesh, and 2D-Torus (cf. Chapter 2 Sub-Section 1.2). The NoC routing component is implemented in this own process.

This approach enables the emulation platform to scale well with the number of emulated clusters without introducing a bottleneck in the SystemC simulation kernel. To prevent a communication bottleneck within the NoC component, we need to adjust the $NoC\_latency$ value according to the number of emulated clusters.

Figure 6-7 depicts the temporal evolution of a NoC synchronization mechanism. It displays a simple case with two clusters. Cluster A sends two commands to cluster B that do not request data. Cluster B sends a data transfer command. The SystemC simulation kernel of cluster A requests more host time than the simulation kernel of cluster B to simulate the $NoC\_latency$ period of SystemC time. This is not an issue since the cluster waits for the router notification to go to the callback phase. After notification, the cluster processes the received commands. Cluster B receives no data request, so it directly sends a callback synchronization header and wait. Cluster A processes the data request from cluster B and sends a callback message encapsulating the data. Then it sends a callback synchronization header and waits. The router forwards the callback message and notifies the clusters when it has received the two synchronization headers. Then, a new run phase is started on each cluster.

This approach based on IPC raises the NoC implementation complexity, but provides emulation multi-threading and so a good scaling of the emulation time with the multiplication of the number of clusters. The only fact to be aware is to have a host machine with the equivalent number of execution threads as the number of emulated clusters.

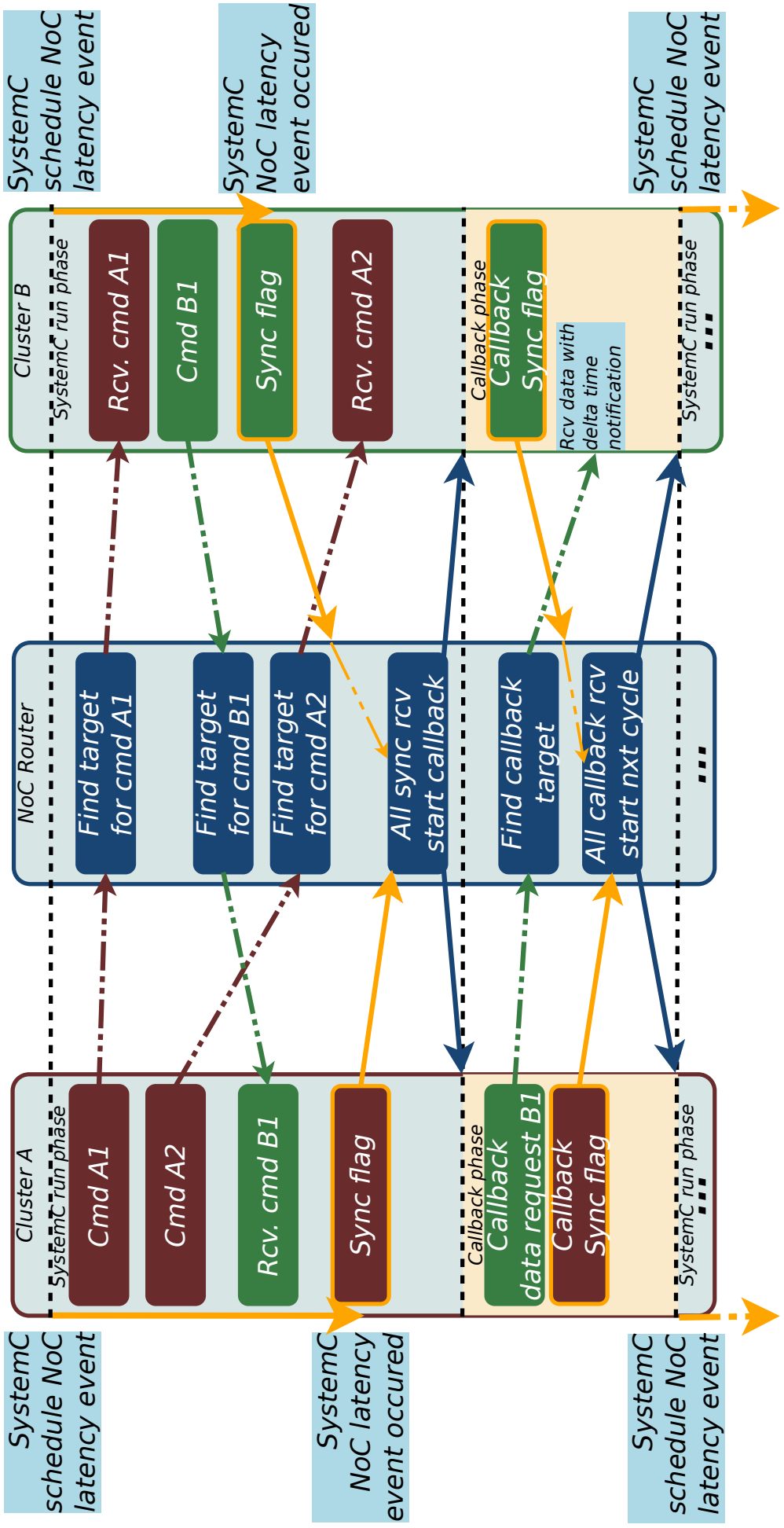


Figure 6-7 – **Overview of the NoC synchronization mechanism.**

2.2-3 Configuration facilities

The previous cluster and NoC structures have been implemented with some hooks to ease the configuration. These hooks enable to change the overall structure of the target HMpSoC with a configuration file. This approach prevents the recompilation of the emulation platform for each new target architecture. Besides, the configuration mechanism from an external configuration file allows the user to customize the emulation architecture according to their needs without digging into the platform emulation source code. And by the way, it extends the number of potential users. Furthermore, this emulation platform has been developed to ease the large scale test of the energy-aware task mapping algorithm. So, the external configuration facilities is mandatory to allow its scripting and its integration as a backend of a DSE framework such as Gecos.

The configuration relies on the textual representation format JavaScript Object Notation (JSON) which is the minimal readable format for structuring data. The configuration file is composed of multiple sections as described on Fig. 6-8 (left side). The global parameter section sets the file path for the IPC socket, the synchronization quantum of QEMU and SystemC and other values such as debug flag, and so on. The NoC parameter section sets the NoC topology, its $x \times y$ size, the maximum packet length and the communication time and energy properties. Then a list of cluster parameters block are fulfiled. The cluster configuration contains also multiple sections as depicted on Fig. 6-8 (right side). The first section sets the cluster address space and the architecture used in QEMU. The second one sets the memory size, its latency and the parameters of each communication channel. The third one sets the GPCom parameters. Then a list of HW IPs loaded in the cluster is given with the target IP and the communication channel parameters.

At the start-up of the emulation platform, this file is parsed and a specific class instantiates each component with the corresponding configuration and manages the TLM socket binding between them. In Appendix B, we display some snippets of each part to highlight the simple syntax used in the configuration file.

These elements provide a flexible HMpSoC structure which is easily configurable. The NoC implementation enables a multi-threaded execution of the emulation platform that eases the execution to scale with a large number of clusters. In fact, the majority of computation server draws their performance from the multiplication of cores, and thus using a mono-threaded emulation platform would have been a major bottleneck toward the emulation complexity scaling. On top of that, the proposed configuration facilities enable to script a bunch of tests on a large number of emulated architectures without having to recompile the emulator or digging into the source code. With these features, this emulation platform is a good candidate to be a test backend for research on mapping algorithms.

2.3 Execution framework and cluster management

The previous section introduced the architectural structure of the emulation platform. This section describes the middleware layer that enables to easily use and program applications over the emulated HMpSoC.

2.3-1 Communication monitoring

As described in Subsub-Section 2.2-1, the cluster implements a communication monitoring feature. All the communications occurring between the processors, the HW accelerators and the shared

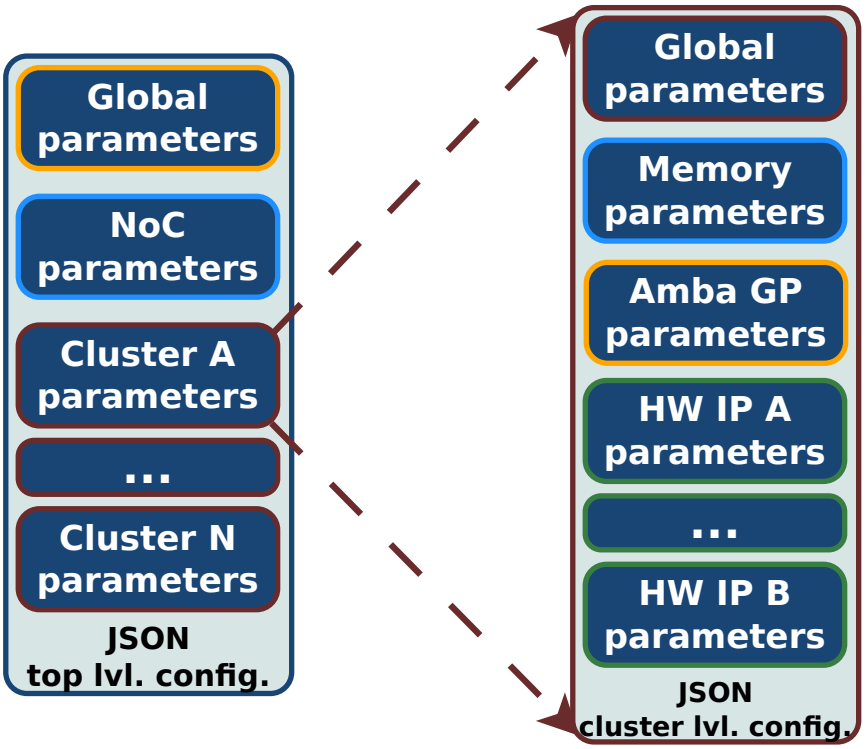


Figure 6-8 – **Overview of the configuration file structure (more details in Appendix B).**

memory bank are monitored. However, the SW part emulated by QEMU embeds a memory space too. This memory is accessed by QEMU with a short latency and could also be accessed by the HW part. One solution would have been to disable it, such that all the memory accesses occur within the cluster memory so that they can be monitored. This approach introduces multiple drawbacks. First, the cluster memory, due to the monitoring feature, presents slower read/write access (in term of host execution time) than the QEMU embedded one. Booting an OS on such a cluster configuration is awfully slow. Furthermore, this approach implies to monitor all the communications involved on the platform such as those initiated by OS system routines and not linked to the application under test.

To address these drawbacks, we decided to keep the internal QEMU memory alongside with the cluster one. The memory used by the OS kernel and subroutine is mapped in the QEMU memory, and all the memory blocks related to the application are mapped in the monitored cluster memory. With this approach, the application memory access reports are not tainted by OS memory accesses that are not directly linked to the tested application. In addition, the OS booting time and the average emulation performance are an order of magnitude better. For this purpose, we need to implement a modified version of the malloc/free functions in order to used only memory block within a specific range corresponding to the monitored memory. The first investigated approach was to use a modified version of *glibc malloc* based on fixed range memory mapped configuration. Unfortunately, the internal implementation of malloc memory management relies on mmap function features that are only available when the target address range is unspecified. So, we decided to keep the standard glibc malloc function and to use a dedicated driver to manage the monitored memory range. This driver manages a bunch of chunks through chained list data and mimics the malloc behavior from a user point of view. To enable this feature in our application code, we first used the malloc hooks to dynamically replace the standard glibc malloc by our custom one. Although this approach correctly works to load our custom malloc function, the reverse operation fails and generates some segmentation faults during the test application unloading. To prevent these errors, we decided to explicitly replace the malloc (and free) calls in the target application by our custom malloc.

The custom malloc driver proposes to the users the same prototypes as standard malloc functions. The memory allocation and release are performed with the help of a pool of memory chunk managed by two linked chain structures. A memory management algorithm is used to find the number of available memory chunks needed. This algorithm is also responsible of memory deallocation and its design to prevent the memory segmentation.

The driver also implements particular features to enable memory block sharing between processes. In fact, multiple tasks could be executed on the same HMpSoC cluster. In order to prevent useless memory copies when those tasks communicate, the custom malloc proposes a zero copy mechanism through the use of memory block sharing. For this purpose, the custom malloc driver implements a subscription mechanism, and the physical memory blocks are released only when all subscribers have asked for memory release.

2.3-2 NoC management

As described in Subsub-Section 2.2-2, the NoC implementation provides low-level message passing mechanisms. In order to be efficiently used, it is necessary to provide a linux driver for management and a higher API to ease its use.

The driver has to manage the link between the NoC task id and the corresponding OS process. It is also responsible of message storing, and it processes wake up management. For this purpose, it keeps a list of all processes that have used the NoC interface alongside with their current status. When an interruption occurs from the NoC for either commands or acknowledgments, the driver finds the corresponding process, stores the event in the process pending event list, and updates its status if necessary. The interruption handling mechanisms use a twofold strategy to maximize the interruption handling capabilities. During the first step, the interruption is acknowledge, the interruption flag is reset and a tasklet is scheduled for being processed later. When possible, the tasklet is executed, it retrieves the corresponding data from the NoC interface and updates the target process data and status.

On the user side, an API is implemented to hide the NoC management complexity to the users. This API provides primitive functions for sending/receiving data to/from a specific node executed on a remote cluster. The API provides non-blocking and blocking version of these functions. With the blocking version, the initiator process is put to sleep with the help of linux standard process handling and then wake-up after the end of the requested operation. The API also provides specific functions that ask to a remote cluster to subscribe to a specific task to cooperatively work. This feature is particularly useful for dynamic loading of application over the HMpSoC as described in the next session.

The combination of the developed driver and the API libraries enables the potential user to efficiently use the full possibility of the NoC with simple high-level function call.

2.3-3 Execution scheme

Each cluster contained in the HMpSoC emulator, runs its own linux OS. The applications are dynamically load on the HMpSoC, thus it is necessary to define and implement a management scheme that relies on the underlying primitive. To allow for the cooperative work of the clusters, one of them is designated as master and the other ones are designated as slaves.

The master cluster starts by parsing the target application graph. For each node in the graph, it issues a work subscription to the target cluster and computes the ancestor and successor node list

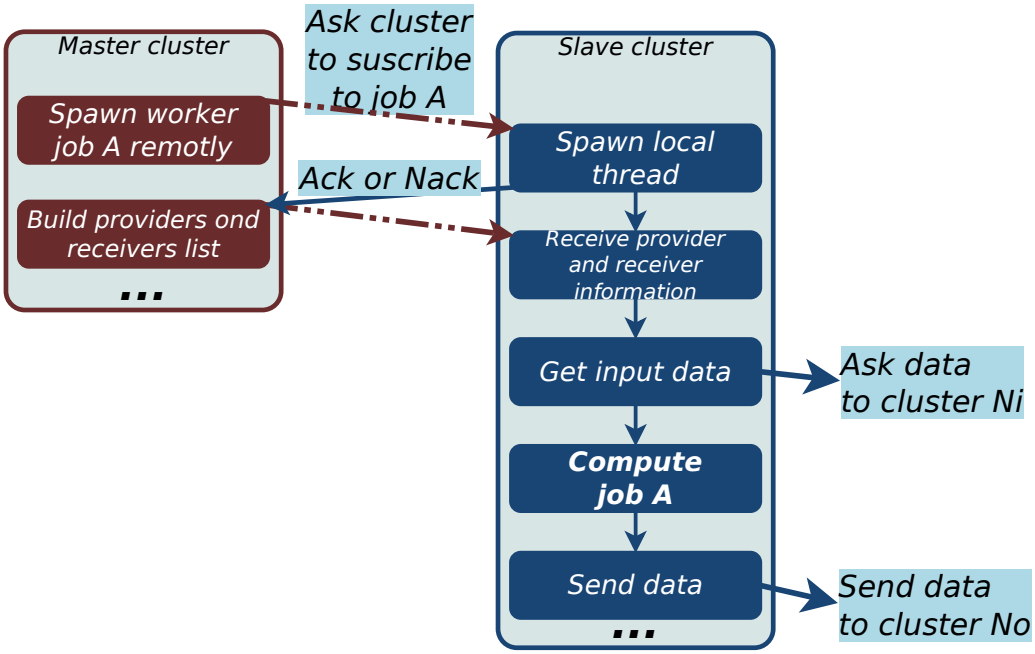


Figure 6-9 – **Remote worker started from master cluster.**

to build the sender and receiver parameters. After that, it sends these values to the target cluster. If the current node needs to be executed on the master cluster, it locally spawns a process and sends him the data through the zero copy local communication mechanisms.

On the other hand, slave clusters register a slave manager process within the NoC interface. This slave manager waits for work subscription from master. After that, the slave manager accepts the work or declines it if it has not the required resources for the work. In case of acceptance, the slave manager spawns a process with the corresponding application stub. The application stub starts and waits for the parameter packet from the master to get the address of sender and receiver nodes for the application data. Then, it retrieves the data, performs the computation, and sends the data to the corresponding nodes.

This mechanism enables the dynamic application broadcasting over the HMpSoC cluster and so the cooperative working of each of them without having to manually load specific application binaries on each one. The execution scheme heavily relies on the NoC communication primitive. Figure 6-9 depicts the remote worker spawning mechanism initiated by the master cluster. The same mechanism is implied for local worker spawning at the difference that only local communications are involved in that case.

3 Application layer

In the previous section, we presented an emulation platform that solves the issue of exploring and testing mapping algorithms on a broad range of architectures. In this section, we address the second issue, which is to be able to test these algorithms on numerous representative applications.

3.1 Representative applications: the dwarf principle

In 2006, researchers from Berkeley propose a survey on the future of parallel computing. This work [Asa+06] investigates the success of parallelism at the extremes of the computing spectrum

from High Performance Computing (HPC) to embedded computing. The involved application structures are analyzed in details, and the authors are surprised by the number of similarities found between embedded computing and scientific computing. Then, they decided to extract the computation and communication patterns of applications and to classify them in generic computation kernels called *"dwarf"*.

A dwarf is a set of computation and data movement properties. Applications that compose these dwarf categories can be implemented with different underlying numerical methods. Nevertheless, the claim is that the underlying patterns have persisted through past generations of changes and will remain important into the future. First, seven dwarf categories are extracted from a set of widespread HPC benchmarks such as NAS (NASA Advanced Supercomputing) [Bai+91]. Then, this dwarf method extraction is applied to a broader range of array computational applications. For this purpose, the previous dwarfs are combined to capture the behavior of complex applications and some new dwarfs are introduced to cover the missing important areas beyond HPC. The fields of machine learning, database software, computer graphics and game are covered. This led to the introduction of six new dwarf categories. On top of that, this study shows that any applications could be defined as a composition of the thirteen obtained dwarfs. By the same way, an infinite set of representative applications could be obtained with the composition of these dwarfs. The application generation mechanism, introduced in Sub-Section 3.3, is based on this observation and use the dwarf implementation from MachSuite benchmark[Rea+14]. Below we give a brief description of the thirteen dwarf categories extracted from [Asa+06]:

- **Dense linear algebra** (e.g. *Gemm blocked*): data are dense matrices or vectors. Such applications use unit-stride memory accesses to read data from rows, and stride accesses to read data from columns.
- **Sparse linear algebra** (e.g. *Spmv*): data sets include many zero values. Data are usually stored in compressed matrices to reduce the storage and bandwidth requirements to access all the non-zero values. Because of the compressed formats, data are generally accessed with indexed loads and stores.
- **Spectral methods** (e.g. *FFT strided*): data are in the frequency domain, as opposed to time or spatial domains. Typically, spectral methods use multiple butterfly stages, which combine multiply-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others.
- **N-body methods** (e.g. *Md knn*): depend on interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others, leading to an $O(N^2)$ calculation, and hierarchical particle methods, which combine forces or potentials from multiple points to reduce the computational complexity to $O(N \times log(N))$ or $O(N)$.
- **Structured grids** (e.g. *Stenc 2D*): represented by a regular grid, points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between two versions of the grid. The grid may be subdivided into finer grids in areas of interest and the transition between granularities may dynamically happen.
- **Unstructured grids** (e.g. *Backprop*): an irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighboring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points, and then loading values from those neighboring points.
- **Monte Carlo** (e.g. *Sort merge*): calculations depend on statistical results of repeated random

trials. Also considered as embarrassingly parallel.

- **Combinational Logic** (e.g. *Aes*): functions that are implemented with logical functions and stored state.
- **Graph traversal** (e.g. *Bfs bulk*): visits many nodes in a graph by following successive edges. These applications typically involve many levels of indirection, and a relatively small amount of computation.
- **Dynamic Programming** (e.g. *Nw*): computes a solution by solving simpler overlapping subproblems. It is particularly useful in optimization problems with a large set of feasible solutions.
- **Backtrack and Branch+Bound**: finds an optimal solution by recursively dividing the feasible region into subdomains, and then pruning subproblems that are suboptimal. This dwarf category is not implemented within the application generator.
- **Construct Graphical Models** (e.g. *Viterbi*): constructs graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models.
- **Finite State Machine** (e.g. *Kmp*): is a system whose behavior is defined by states, transitions defined by inputs and the current state, and events associated with transitions or states.

3.2 Dwarfs implementation within the execution structure

We decided to use this dwarf principle to generate representative applications instead of porting a generic benchmark application set on the simulator. In fact, if a version of each dwarf is available on the emulator platform we can approximate the behavior of most applications by a simple composition of them. To ease the implementation of these dwarfs, we used as a staring point the implementation proposed in machSuite [Rea+14] and implemented twelves distinct dwarf categories on the emulation platform.

3.2-1 Wrapper structure and communication scheme

The machSuite [Rea+14] implementations first target the evaluation of HLS tools and accelerator-centric architectures. Hence, they use small input size. To mitigate this phenomenon and generate configurable dwarfs input size, we extended the kernel of each dwarf with an outer loop that control the number of kernel iterations. To ease the dwarf spawning across the emulation platform, we developed a generic wrapper around the dwarf kernel. This wrapper gathers the input data in one array, and the output data in another one. To manage the case of dwarf generating in-place results (result are written in the memory slot of input data), the wrapper was extended with a Boolean value that stipulates when the dwarf has generated in-place results and when it has allocated memory. These wrappers are then injected into a job management process that gathers the input data from the different clusters and spreads the output results to different clusters after the computation. With the iteration mechanism and the proposed encapsulation, each dwarf represents a *BpB* in our previously introduced application structure (cf. Chapter 2 Section 4).

These encapsulation and communication facilities enable to generate applications composed of a set of dwarfs with a complex inter-task communication scheme. These wrappers are then gathered in a map structure and thus can be chosen at execution time following the command issued by the master cluster.

3.2-2 SW implementations

To match with the BpB definition that requires the data reading in head and output writing in queue, each dwarf uses local buffer for input and output data. The computation kernel is inserted in a loop that manages iteration over kernel and thus enables to change the number of input data required as well as the output generated or the global amount of computation.

3.2-3 HW implementations

The previous dwarf implementations are inserted in a SystemC class that inherits from a generic HW IP structure. This structure defines the number of registers that configures the IP. The wrapper around the dwarf implementation lets them being managed by the same mechanism and with the same number of configuration registers. The generic IP structure integrates a mechanism to monitor the time and energy needed by the execution of the IP. This mechanism is based on two values that are set up in the configuration file loaded at run-time (cf. Subsub-Section 2.2-3).

On top of that, a generic IP driver is introduced. It exposes the required primitive to manage efficiently the HW IP of each dwarf. This driver configures the IP registers. The,n it starts the computation and waits for a HW interruption to wake-up and go ahead.

3.3 Generic graph generation

Our main objective is to generate applications composed of a random composition of dwarfs with random communication pattern between them. This random application generation needs to be configured by the user to obtain an application graph adapted to his need. Namely, the user needs to choose: the wideness of the graph which is correlated to the parallelism degree of the obtained application, the connection density between the dwarfs which influences the obtained dependency and communication patterns, and so on. To help the user through this configuration step, we proposed a GUI that is detailed below.

3.3-1 Generator structure

Figure 6-10 displays an annotated version of the GUI. As we can see, the control panel is composed of four main parts. The first one (frame **A**) is the actions panel, from the available buttons the user drives the call of the underlying methods for generating new graph, saving it in binary format and loading old one. The second part (frame **B**) defines the global shape of the desired graph. The width of the graph is strongly connected to the available parallelism degree in the obtained application. The depth and the number of nodes influence the size of the generated application. Lastly, the density and edge length influence the obtained communication patterns. The third part (frame **C**) manages general parameters. The user can define the number of available clusters, the maximum number of dwarf iterations as well as general properties such as node properties showing and random mapping generation. The last part (**D**) enables the user to select the used dwarf categories.

The rest of the GUI is used to display the generated graph (frame **E**). On top of each node, a small frame (**F**) depicts their properties such as the used dwarfs, the implementation type (HW or SW), the target execution cluster and the number of iterations. These properties could be edited by clicking

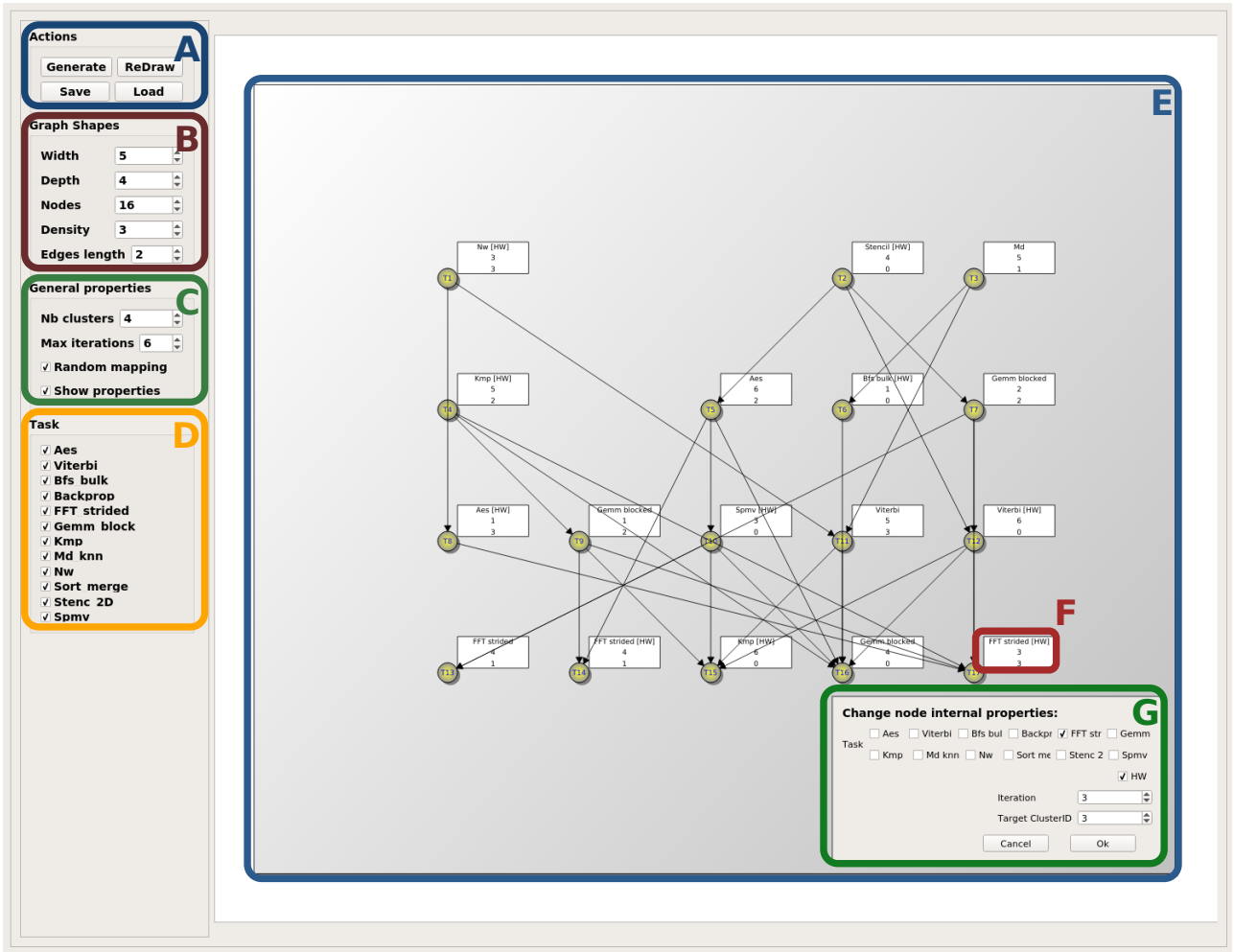


Figure 6-10 – **Overview of the graph generator GUI.**

on the node, action that pops a small configuration window (frame **G**) in which each property could be edited by the user.

This GUI is written in C++, with the help QT, and could be easily extended to future needs.

3.3-2 Graph execution and monitoring facilities

Once graphs are generated with the desired properties and saved in binary format, we need to provide facilities to execute them on the HMpSoC. To this aim, two solutions are possible. The first one consists in providing a translator that converts the graph structure into an application source code for the master cluster. The second solution is to provide an application stub for the master slave that dynamically parses the graph structure and spreads the work on the HMpSoC as intended by the mapping rules.

The first solution requires the compilation of the obtained source code for each new application. This implies that the user has set up a full cross-compilation tool-chain targeting the ISA running on the HMpSoC and prevents the distribution of pre-built binaries. Our aim is to ease the use of the developed HMpSoC emulation platform, for this reason, we left aside this solution and favored the second one. The ease of use of the second solution comes with a small amount of extra computations at run time but ,in regard of the size of each dwarf, this is not an issue. The dynamic graph parsing application reads the graph with a bottom-up approach. This guaranties that all the target nodes of

```
"hwCpn": {
    "aes_t1": {...},
    "viterbi_t2": {...},
    "bfs_bulk_t3": {...},
    "backprop_t4": {...},
    "fft_strided_t5": {...},
    "gemm_blocked_t6": {...}
}
```

(a) **Configuration cluster type A**

```
"hwCpn": {
    "kmp_t7": {...},
    "md_knn_t8": {...},
    "nw_t9": {...},
    "sort_merge_tA": {...},
    "stenc2d_tB": {...},
    "spmv_crs_tC": {...}
}
```

(b) **Configuration cluster type B**

Figure 6-11 – **HW accelerators embedded in the test clusters.**

a current executed node is already registered when it finishes its computation. For each node, the dynamic graph parser computes the ancestor node list and the successor one to infer the desired communication pattern between the nodes. All nodes wait until their dependencies are satisfied, then they start their computation. The synchronization mechanism relies on the primitives provided by the NoC structure and the local communication facilities.

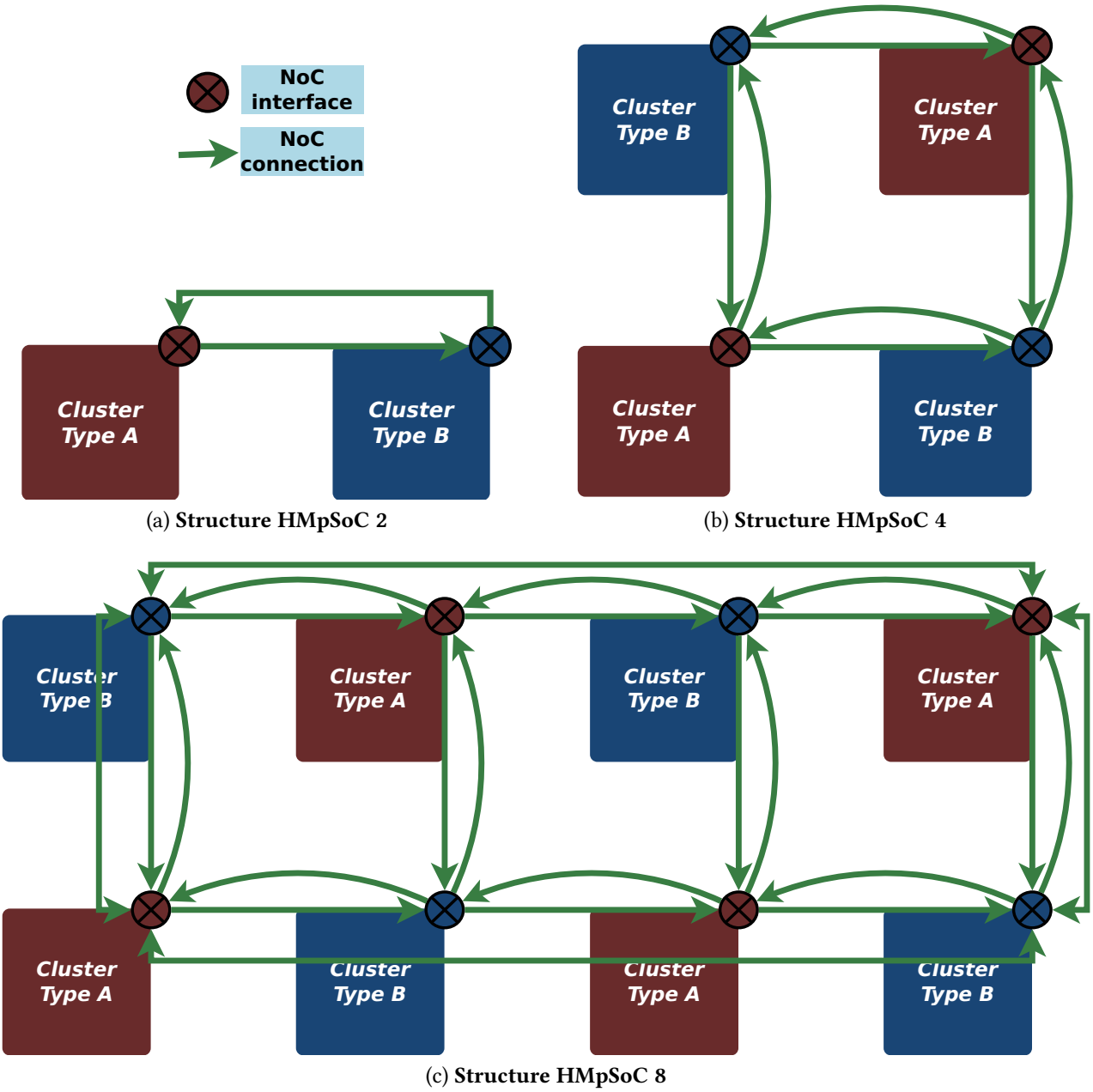
On top of that, the dynamic graph parser manages the power consumption monitoring feature. It resets the monitoring structure at the beginning and at the end of execution displays to the user their values. In the next section we show the embedded power measure results for a set of applications and HMpSoC configurations.

3.4 Communication energy monitoring results

In this section we validate the implementation of the communication-based power model within the HMpSoC emulator as well as the parameter configuration facilities. For this purpose, we build three different HMpSoC configurations composed of two, four, and height clusters respectively. Each cluster is composed of two ARM processors and six HW accelerators. We built two HW accelerator combinations that led to two cluster types (cf. Fig. 6-11).

The NoC structure used in the two-cluster configuration is a *RING*, a *MESH* structure is used in the four-cluster configuration and a *2D-TORUS* is used in the height-cluster configuration. The obtained HMpSoC overall structures are depicted on Fig. 6-12.

Then, for each architecture, we generate three random application graphs with various properties. Each of them is detailed in Appendix C. These graphs are then executed on the HMpSoC and the communication monitoring results are compared to the expected value from the communication-based power model introduced in Chapter 4. The obtained results are displayed in Table 6-1. In this table, the number of BpB contained in each application graph is displayed with the estimation results of an offline method (*estimated*) and with the communication monitoring results (*simulated*). The relative error between the online and the offline results is also given. This error is between $2\,\%$ and $6\,\%$ for the energy metric and is under $1\,\%$ for the time metric. This variance is mainly due to the inaccuracy of the offline estimation method, which does not take control communications into account for complexity reasons. These control communications are generated by the master cluster when it spreads the work over the HMpSoC, as well as by each cluster when they set up HW accelerators. The zero-copy mechanism involved within the local communication, also generates some control communications that could not be managed by the offline estimation. Unlike the offline estimation, the monitoring method embedded in the emulation platform can consider these control communications and thus can provide a more precise energy consumption estimation.



(a) **Structure HMpSoC 2**

(b) **Structure HMpSoC 4**

(c) **Structure HMpSoC 8**

Figure 6-12 – **Overview of the HMpSoC architectures used for the tests.**

4 Conclusion

To circumvent the drawbacks implied by testing the DSE method on multiple architectures and applications, we decided to built a HMpSoC emulator dedicated to the test of mapping algorithms under energy constraints, which is described in this chapter.

First, we have introduced the characteristics of the used technologies and the structure of the designed HMpSoC alongside with the configuration layer that enables to target a large panel of HMpSoC architectures. In order to be efficiently used in the field of energy-aware mapping algorithm, this emulator was developed in association with a representative application generator. The application generator relies on a set of dwarf application kernels randomly associated to obtain full-flavor applications.

The HMpSoC emulator is delivered with a full application middleware that enables to manage

		Energy [J]			Time [s]		
	Nb. BpB	estimated	simulated	error [%]	estimated	simulated	error [%]
arch. 2	6	0.01968	0.02095	6.084	1.71329	1.71445	0.067
	7	0.00715	0.00738	3.164	0.32301	0.32441	0.430
	8	0.01517	0.01546	1.821	0.98136	0.98924	0.797
arch. 4	15	0.01269	0.01351	6.092	0.42703	0.42891	0.438
	15	0.01739	0.01814	4.141	0.64357	0.64618	0.404
	15	0.01071	0.01108	3.347	0.57534	0.57741	0.358
arch. 8	26	0.02026	0.02128	4.820	0.79467	0.79503	0.045
	20	0.03581	0.03785	5.390	0.47235	0.47261	0.055
	20	0.02546	0.02623	2.950	0.63333	0.63354	0.034

Table 6-1 – **Monitoring results of communication energy and time.**

the low-level components and to execute the generated applications without external intervention. On top of that, the emulator embeds the previously introduced communication-based power model as well as a complete set of binding to integrate an energy estimation of the computation part. This emulation platform, as well as the application middleware, are released under the open-source MIT license and could be retrieved from the GitHub platform (emu-HMpSoC[1]).

The energy communication monitoring has been tested with three distinct HMpSoC architectures on nine application graphs. As depicted on Table 6-1, the results are accurate. However, some features are still missing to make this simulator fully functional for the validation of energy-aware mapping algorithms. These missing features are presented below.

Currently, the energy monitoring of communications is fully functional within the emulator. However, we need to implement a power model for the computation of each tasks. For the HW tasks, the power consumption should be extracted from a computation cost parameter embedded in the configuration file and correlated with the number of iterations needed by the algorithm to converge to the solution. In addition, the impact of the convergence iteration number should be mitigated with another cost parameters when the underlying HW architecture is pipelined. In that case, these two cost factors will represent the energy dissipated during setup iteration (HW latency) and during the established one (Initiation Interval). The HMpSoC emulator infrastructure already contains the binding for this, but the dwarf application stub needs to be adapted to compute the impact of the HW pipelining.

For the SW side, the power model implementation will be slightly different and could rely on the ILPA method (cf. Chapter 3 Subsub-Section 1.2-2). For this purpose, one implementation method will be to compile the dwarf application kernels with an extra compilation step that increments some counters for each instruction category and then periodically updates the global cluster energy counter by associating the instruction count with their costs.

[1] `https://github.com/rouxb/emu-HMpSoC`

Chapter **SEVEN**

CONCLUSION & PERSPECTIVES

In this chapter, the global context of the work is summarized and an overview of the contributions proposed in this thesis is recalled. Mid- and long-term perspectives opened up by this work are also given.

1 Conclusion

In the last decade, the design of embedded systems was pushed to increase computational power while maintaining low energy consumption. With the advent of Multiprocessor System-on-Chip (MpSoC) architectures, simplification of processor cores decreased the power consumption per operation, while the multiplication of cores brought performance improvements. However, the *dark silicon* issue led to add specialized hardware (HW) accelerators to programmable processors and to the rise of Heterogeneous MpSoC (HMpSoC) architectures, which combine both software (SW) and hardware computational resources. HMpSoC turned out to be good candidates for implementing Software-Defined Radio (SDR) applications. Indeed, they bring high computation performance, high reconfiguration facilities with good power efficiency whereas the application design complexity increases. Within this thesis, we proposed new methodologies and tools that help the application designer to achieve power efficiency when addressing this new kind of complex architectures.

Firstly, we introduced the global context of this work. SDR application characteristics and their involved computation patterns were depicted alongside with the new trends, such as the "Dark Silicon", that govern the computer architecture and semiconductor technology domains. A panel of architectures used within the SDR field was introduced as well as some general purpose manycore architectures. These observations led us to the formulation of a generic definition of the HMpSoC architecture families targeted in this work. We also proposed a HMpSoC representation model focusing on communication channels. This representation gave a precise macro view of the architecture structure and was used for HW/SW partitioning and task mapping afterthought. In addition, we presented some application representation paradigm with their relevant properties and we aggregated them to build the EPDG representation that was used in this work. Once the context and motivation of this thesis was clearly defined, we presented the relevant related work. The existing power modelling methods were presented with their advantages and limitations. We also exposed the various available CAD tools that already help application designer to extract application parallelisms in order to exploit the numerous computation cores of HMpSoC. Similarly, we presented some DSE methodologies and tools that help the designer to make choices during the various design steps.

Then, we proposed a coarse-grain energy-aware design flow structure that targets HMpSoC and we extracted the main requirements, in terms of methods and tools, needed. The previous observations have shown that HMpSoC performance and energy consumption depend on a large set of

parameters, such as the HW/SW partitioning, the type of HW implementation or the communication cost. Besides, the intrinsic structures of these architectures exposed a large set of options for each of these parameters, thus leading to a huge design space. When energy consumption is a key requirement of the application, this design space must be explored. However, due to the design space size, the state-of-art power modelling methodologies are not applicable due to their computation time.

To circumvent this issue, we proposed a fast power modelling approach focusing on communications occurring across the computation cores. For this purpose, we started by highlighting the communication effects on two commercially available manycores or heterogeneous architectures with the help of micro-benchmarks. This enabled us to extract the communication cost of each communication channel. We then proposed the power modelling formulation that independently considers the influence of each power consumption source and therefore leads to fast estimation within the HW/SW partitioning and task mapping steps. We validated this formulation through synthetic applications that generate random communication patterns across the Zynq architecture. These experiments have shown that the achieved estimation accuracy is largely enough to be used in the task mapping step, while the estimation time also fits design space exploration constraints.

Using this power modelling facilities we proposed an energy-driven accelerator exploration for HMpSoC. This DSE method targets kernel-based applications and helps the user to select the optimal implementation among the available ones with the best task distribution across the cores. The starting point of this method is to extract the computation cost of each available implementation (SW or HW). For this purpose, we used a micro-benchmarking approach, but other estimation tools could also be used. Then, we merged these results with the previous extracted communication cost within an MILP formulation. This formulation depicts the set of constraints of the targeted application and architecture alongside with cost functions. The use of an MILP solver enables to extract the optimal solution within a second. This methodology was tested on two application kernels: a matrix multiplication and a stencil computation on a Zynq-based heterogeneous architecture. These experiments showed that the acceleration and energy saving obtained are at minimum of $12\,\%$ compared to solution obtained with more conventional approaches. We also showed the adequacy between the estimated and achieved results . The estimation error is within $5\,\%$ to $10\,\%$, which is mainly due to the sensor inaccuracy.

These results opened up new opportunities for future CAD tools. However, this work also highlighted some difficulties linked to the test on a broad range of applications and targeted architectures. These difficulties limit the validation and the development of new energy-aware HW/SW partitioning and mapping algorithms. To circumvent this issue, we proposed a HMpSoC emulation platform. This platform targets the evaluation of HW/SW partitioning and mapping algorithms at a large scale. To this purpose, the platform is highly configurable through simple configuration file. It provides an integrated communication monitoring and energy estimation facilities to build a global energy estimation model on top of them. It also keeps a good simulation time through the use of QEMU and SystemC technologies. On top of that, the simulator is released with an integrated application and execution framework that aims to automatize the validation process with simple configuration scripts. The structure of the emulation platform was precisely described and the source code of the emulator is released on the GitHub platform.

Although the emulator platform is functional, some improvements need to be included to efficiently address the HW/SW partitioning and mapping algorithms research field. These improvements are detailed in the next section as the perspectives of this work.

2 Perspectives

The HMpSoC architectures are at the leading edge of embedded computing for the next decade. In this PhD thesis, we have explored the opportunities and needs in terms of tools and methods to address the new challenges proposed by this kind of architectures. However, some improvements remain to be brought to the proposed tools and different long-term perspectives arise.

First, the validation process of the tool needs to be improved. In fact, the validation of the emulation platform in terms of energy estimation was limited to the communication cost between the computation cores. The implementation of the energy estimation for the computation part has to be finalized. The integration of an Instruction Level Power Analysis (ILPA) modelling, through the help of a dedicated compilation step, should be investigated for the SW part. For the HW side, the emulator already exposes two parameters with the corresponding hook points to estimate the energy consumption. However, these estimation processes should be tweaked and validated as well as the overall estimation using the proposed application generator.

On top of that, the proposed DSE method based on MILP formulation was only applied on two regular computation kernels. It will be interesting to evaluate the performance of the method on each dwarf category. For the dwarfs that can be tiled, the MILP formulation could be used with no modifications. For the other dwarfs, the MILP formulation has to be extended with a preliminary step and other tricks to manage the irregular inter-task dependencies. However, the various parallelism extraction tools should be able to split these dwarf applications in a set of parallel tasks and so to achieve a similar structure as the tiled application kernel, that could be injected in the MILP model.

Once this extension done, the MILP exploration could be further extended from kernel level to full application level. One solution is to insert the MILP formulation as an optimization kernel within a greedy exploration algorithm or within another inclusive MILP formulation and thus address the optimization of a full application generated with a combination of dwarfs. The MILP formulation will generate the Pareto front following the energy cost and the resources used for each dwarf used in the application. The obtained set of points could be then aggregated to obtain the global configuration that achieves the lowest energy consumption for the full application and thus leads to complete energy-aware exploration algorithms.

The proposed emulation platform will show its usefulness by enabling a large scale test of the exploration algorithm. The scripting capabilities of the emulation platform enables the user to test the exploration algorithms on a representative panel of HMpSoC architectures and applications at a low cost in term of development time. This large-scale test will allow to quickly find the limitations of the algorithms and thus their improvements.

Furthermore, once the exploration algorithm performance has been demonstrated, the algorithms could be integrated in existing source-to-source frameworks, such as Gecos, as a standalone optimization pass. In addition, the emulation platform could also be integrated in this kind of tools as an execution test backend that could quickly estimate the execution cost of a given application on a given architecture. The association of the algorithm and the execution backend integrated in the same framework will provide the user with an end-to-end solution to easily implement energy-efficient applications. This will represent a real advance in CAD tools and will strongly ease the large adoption of HMpSoC architectures.

However, these tools will only propose offline optimization for applications that do not change over the time following external parameters. Taking back into consideration the SDR application field, and considering not only the computation and communication patterns of these applications

but their working environment globally, dynamic changes can occur over the time. In fact, for a given SDR communication standard, different execution modes are available. These modes could be dynamically swapped to adapt the wireless communication system to the communication channel changes (e.g. the air interface). These dynamic changes could not be managed by the previous proposed offline optimization. To circumvent this drawback, we could imagine new mixed DSE tools that will associate offline pre-optimization with an online manager. The manager will be in charge of selecting the adapted configuration among a set of them to match with current air interface parameters. However, this requires to compute and embed a dedicated configuration for each air interface state and wireless communication mode and thus leads to a large configuration set. This approach also implies a lot of offline computations and the storing within the embedded communication system memory of a set of pre-computed configurations. Another solution is to give more flexibility to the online manager and to only embed partial optimal configurations that could be merged at run-time to obtain the right solution. This approach will require more online intelligence and computations. Finding the right compromise between these solutions will be a real challenge and represents an interesting long-term perspective for this research field.

Part I

Appendices

Appendix A

MANUAL LOOP-NEST MODIFICATION

```
#Include "ker_matmult.h"
void ker_matmult_0x114(float buffer_A[TILE_SIZE][MTX_SIZE],
                       float buffer_B[MTX_SIZE][TILE_SIZE],
                       float buffer_OUT[TILE_SIZE][TILE_SIZE]){
  lnPixx: for (int i = 0; i < TILE_SIZE; i ++){
    lnPxjx: for (int j = 0; j < TILE_SIZE; j ++){
        float tmpAcc=0;
      lnPxxk: for (int k = 0; k < MTX_SIZE/4; k += 1){
#pragma HLS PIPELINE
          float tmpk0 = buffer_A[i][(4*k)] * buffer_B[(4*k)][j];
          float tmpk1 = buffer_A[i][(4*k)+1] * buffer_B[(4*k)+1][j];
          float tmpkm01 = tmpk0 + tmpk1;
          float tmpk2 = buffer_A[i][(4*k)+2] * buffer_B[(4*k)+2][j];
          float tmpk3 = buffer_A[i][(4*k)+3] * buffer_B[(4*k)+3][j];
          float tmpkm23 = tmpk2 + tmpk3;
          float tmpkm03 = tmpkm01 + tmpkm23;
          tmpAcc+= tmpkm03;
        }
        buffer_Out[_2D_TILE(i,j)] = tmpAcc;
      }
    }
  return;
}
```

Algorithm 4: Loop nest for Impl. LnP 114

```
#include "ker_matmult.h"
void ker_matmult_0x148(float buffer_A[TILE_SIZE][MTX_SIZE],
                       float buffer_B[MTX_SIZE][TILE_SIZE],
                       float buffer_OUT[TILE_SIZE][TILE_SIZE]){
#pragma HLS ARRAY_PARTITION variable=buffer_A cyclic factor=4 dim=1
#pragma HLS ARRAY_PARTITION variable=buffer_B cyclic factor=4 dim=1
  lnPixx: for (int i = 0; i < MTX_TILE; i ++){
    lnPxjx: for (int j = 0; j < MTX_TILE/2; j ++){
        float j0tmpAcc=0;
        float j1tmpAcc=0;
      lnPxxk: for (int k = 0; k < MTX_SIZE/8; k += 1){
#pragma HLS PIPELINE
          // tile_j 0
          float j0k0 = buffer_A[i][(8*k)]   * buffer_B[(8*k)]  [(2*j)];
          float j0k1 = buffer_A[i][(8*k)+1] * buffer_B[(8*k)+1][(2*j)];
          float j0km01 = j0k0 + j0k1;
          float j0k2 = buffer_A[i][(8*k)+2] * buffer_B[(8*k)+2][(2*j)];
          float j0k3 = buffer_A[i][(8*k)+3] * buffer_B[(8*k)+3][(2*j)];
          float j0km23 = j0k2 + j0k3;
          float j0km03 = j0km01+j0km23;

          float j0k4 = buffer_A[i][(8*k)+4] * buffer_B[(8*k)+4][(2*j)];
          float j0k5 = buffer_A[i][(8*k)+5] * buffer_B[(8*k)+5][(2*j)];
          float j0km45 = j0k4 + j0k5;
          float j0k6 = buffer_A[i][(8*k)+6] * buffer_B[(8*k)+6][(2*j)];
          float j0k7 = buffer_A[i][(8*k)+7] * buffer_B[(8*k)+7][(2*j)];
          float j0km67 = j0k6 + j0k7;
          float j0km47 = j0km45+j0km67;

          float j0km07 = j0km03+j0km47;
          j0tmpAcc += j0km07;

          // tile_j 1
          float j1k0 = buffer_A[i][(8*k)]   * buffer_B[(8*k)]  [(2*j)+1];
          float j1k1 = buffer_A[i][(8*k)+1] * buffer_B[(8*k)+1][(2*j)+1];
          float j1km01 = j1k0 + j1k1;
          float j1k2 = buffer_A[i][(8*k)+2] * buffer_B[(8*k)+2][(2*j)+1];
          float j1k3 = buffer_A[i][(8*k)+3] * buffer_B[(8*k)+3][(2*j)+1];
          float j1km23 = j1k2 + j1k3;
          float j1km03 = j1km01+j1km23;

          float j1k4 = buffer_A[i][(8*k)+4] * buffer_B[(8*k)+4][(2*j)+1];
          float j1k5 = buffer_A[i][(8*k)+5] * buffer_B[(8*k)+5][(2*j)+1];
          float j1km45 = j1k4 + j1k5;
          float j1k6 = buffer_A[i][(8*k)+6] * buffer_B[(8*k)+6][(2*j)+1];
          float j1k7 = buffer_A[i][(8*k)+7] * buffer_B[(8*k)+7][(2*j)+1];
          float j1km67 = j1k6 + j1k7;
          float j1km47 = j1km45+j1km67;

          float j1km07 = j1km03+j1km47;
          j1tmpAcc += j1km07;
        }
        buffer_Out[i][(2*j)] = j0tmpAcc;
        buffer_Out[i][(2*j)+1] = j1tmpAcc;
      }
    }
  return;
}
```

Algorithm 5: Loop nest for Impl. LnP 148

Appendix B

HMPSOC CONFIGURATION FILE

```
{
    "global":{
        "sk_bpath": "/tmp/emu-hmpsoc-tmp",
        "sk_qemu": "qemu-rport-_cosim@",
        "sk_noc": "nocIpc",
        "ssh_port_offset": 10300,
        "gdb_port_offset": 9000,
        "sync_quantum": 1000000,
        "sync_icount": 7,
        "preBoot_ms": 700,
        "multiThread_sim": true,
        "run_trace": false,
        "fast_dbg": true
    },
    "noc":{
        "topology": "RING",
        "xSize": 2,
        "ySize": 1,
        "params": {
            "maxPacketLength": 512,
            "tns_atm": 50,
            "tns_rcc": 10,
            "tns_sbc": 1,
            "enj_atm": 0.05,
            "enj_rcc": 0.01,
            "enj_sbc": 0.001
        }
    },
    {...} clusterConf {...}
}
```

Listing B.1 – **Example of global properties and NoC properties configuration options.**

```
{
    "global":{...},
    "noc":{...},
    "clusters": {
        "MasterCl": {
            "gb_addr": "0x60000000",
            "ms_addr": "0x5fff0000",
            "gt_addr": "0x5ffef000",
            "noc_addr": "0x5ffe0000",
            "x_pos":  0,
            "y_pos":  0,
            "cpu": {
                "dtb_name": "emu-hmpsoc_t1_6.dtb"
            },
            "memory": {
                "addr_offset": "0x40000000",
                "addr_space": "0x1f000000",
                "latency_ns": 500,
                "HPx": 6,
                "comChan": {
                    "cpu": {
                        "maxBurst": 300,
                        "engS": 0.03,
                        "engD": 0.003,
                        "timeS_ns": 300,
                        "timeD_ns": 30
                    },
                    "HPx_1": {...},
                    "HPx_2": {...},
                    {...}
                }

            },
            "ambaGP": {
                "addr_offset": "0x5f000000",
                "addr_space":  "0xfdffff",
                "comChan": {
                    "cpu": {
                        "maxBurst": 20,
                        "engS": 0.5,
                        "engD": 0.20000000298023225,
                        "timeS_ns": 50,
                        "timeD_ns": 20
                    }
                },
                "hwCpn": {...}
            }
        },

        "slaveCluster": {...}
        }
    }
}
```

Listing B.2 – **Example of cluster properties configuration options.**

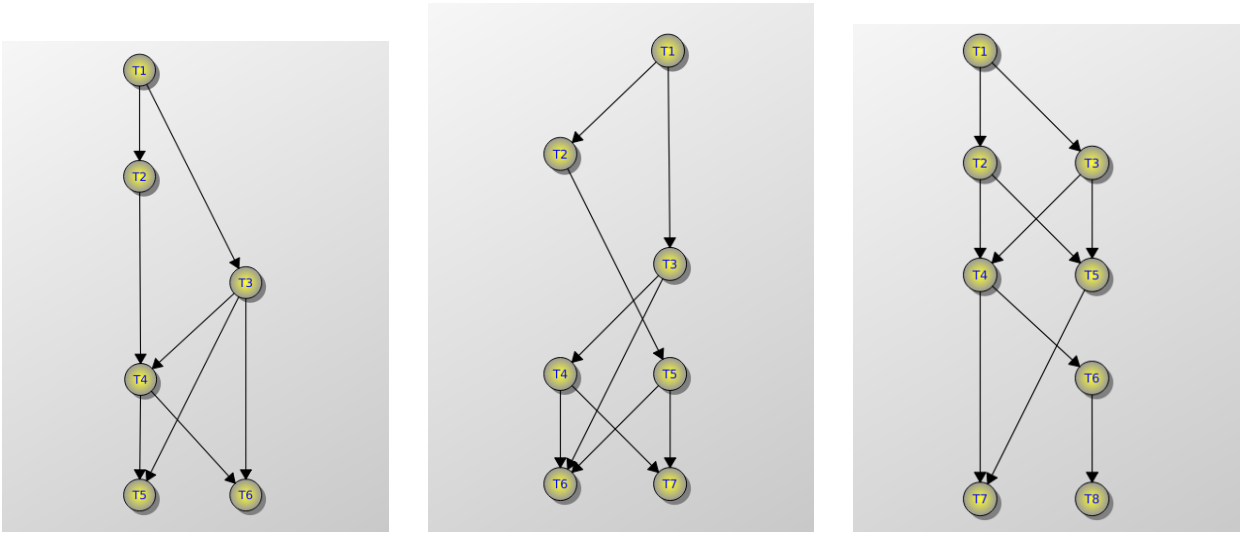
```
"hwCpn": {
    "aes_t1": {
        "ipType": "aes",
        "hpx_id": 0,
        "irq_id": 0,
        "addr_offset": "0x0100",
        "addr_space": "0x40",
        "compTime_ns": 10000,
        "compEn_nj": 100000,
        "comChan": {
            "amba": {
                "maxBurst": 60,
                "engS": 0.06,
                "engD": 0.006,
                "timeS_ns": 600,
                "timeD_ns": 60
            }
        }
    },
    "viterbi_t2": {...},
    "bfs_bulk_t3": {...},
    ...
}
```

Listing B.3 – **Example of HW IP properties configuration options.**

Appendix C

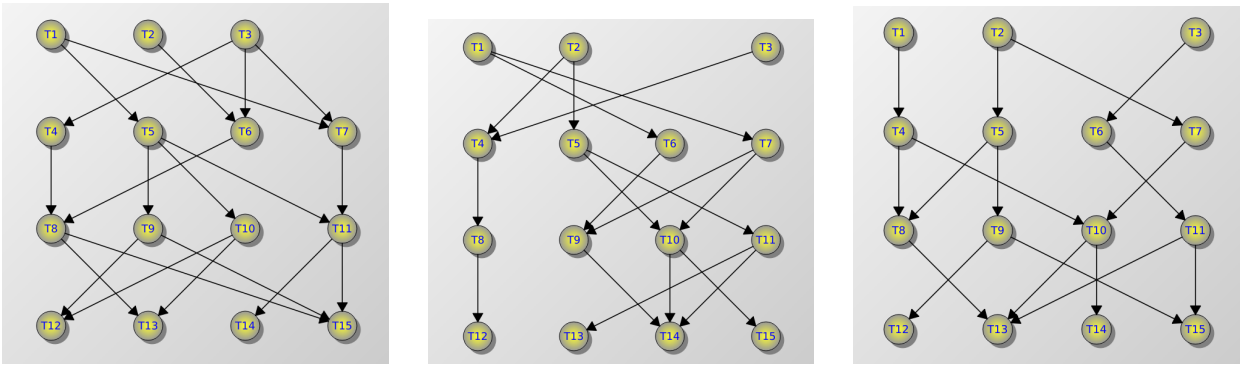
GENERATED APPLICATIONS GRAPH

Below we display the application graphs used for evaluating the communication monitoring of the HMpSoC emulator. These application graph target three HMpSoC architecture composed of two, four and height cluster (cf. Sub-Section 3.4).



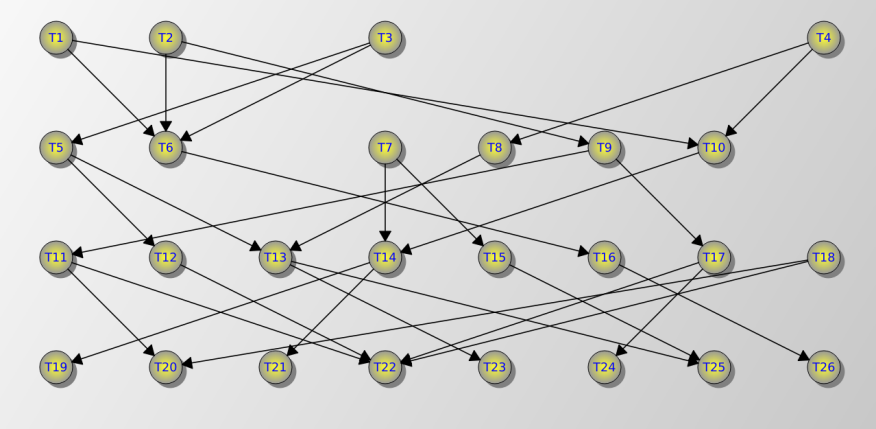
(a) **Application 1** (b) **Application 2** (c) **Application 3**

Figure C-1 – **Overview of the application graph structures targeting the two clusters architecture.**

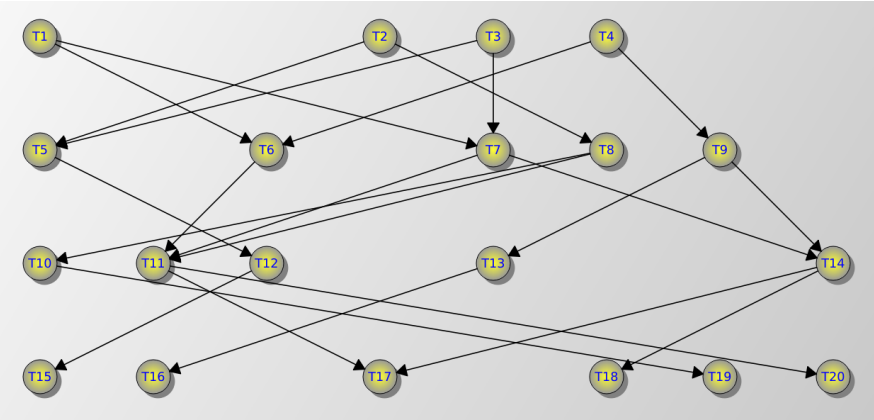


(a) **Application 1** (b) **Application 2** (c) **Application 3**

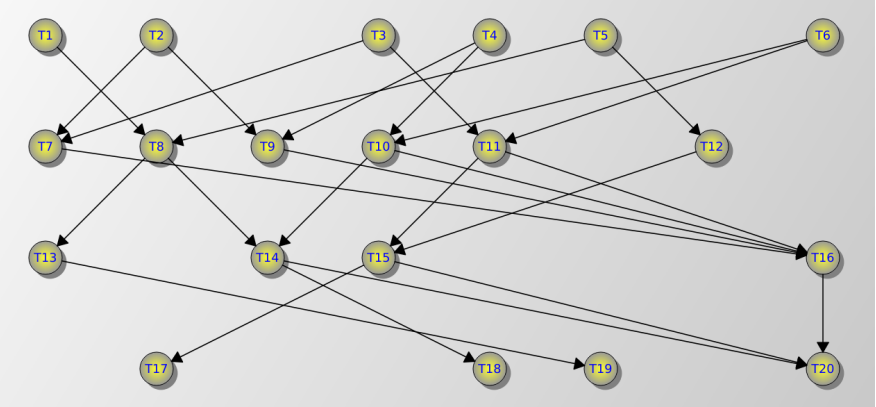
Figure C-2 – **Overview of the application graph structures targeting the two clusters architecture.**



(a) **Application 1**



(b) **Application 2**



(c) **Application 3**

Figure C-3 – **Overview of the application graph structures targeting the height clusters architecture.**

PUBLICATIONS

International Conferences

[Rou+16] Baptiste Roux, Matthieu Gautier, Olivier Sentieys, and Steven Derrien. “Communication-Based Power Modelling for Heterogeneous Multiprocessor Architectures”. In: *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (mc-SoC)*. Sept. 2016.

[Rou+17] Baptiste Roux, Matthieu Gautier, Olivier Sentieys, and Jean-Philippe Delahaye. “"Fast and Energy-Driven Design Space Exploration for Heterogeneous Architectures"”. In: *IEEE 25th International Symposium on Field-Programmable Custom Computing Machines (FCCM), Poster*. May 2017.

BIBLIOGRAPHY

[04] *Digital Video Broadcasting (DVB) : Second generation framing structure, chan- nel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications*. Ref. ETSI EN 302 307 v1.1.1. June 2004.

[05] *Digital Video Broadcasting (DVB) : User guiderlines for the second generation system for Broadcasting, Interactive Services, News Gathering and other broad- band satellite applications (DVB-S2)*. Ref. ETSI TR 102 376 v1.1.1. Feb. 2005.

[97] *Digital Video Broadcasting (DVB) : Framing structure, channel coding and mo- dulation for 11/12 GHz satellite services*. Ref. ETSI EN 300 421 v1.1.2. Aug. 1997.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling". In: *Computer* (Feb. 2002).

[ALG00] Bogliolo Alessandro, Benini Luca, and De Micheli Giovanni. "Regression-based rtl power modeling". In: *ACM Transactions on Design Automation of Electronic Systems*. July 2000.

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Spring Joint Computer Conference*. Apr. 1967.

[Ami+12] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. "Par4All: From Convex Array Regions to Heterogeneous Computing". In: *International Workshop on Polyhedral Compilation Techniques HiPEAC*. Jan. 2012.

[Amm+16] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid. "On Exploiting Energy-Aware Scheduling Algorithms for MDE-Based Design Space Exploration of MP2SoC". In: *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Feb. 2016.

[And+10] Charles André, Julien DeAntoni, Frédéric Mallet, and Robert De Simone. "The Time Model of Logical Clocks available in the OMG MARTE profile". In: *Synthesis of Embedded Software*. June 2010.

[AS15] Manikandan SK Aiswarya S and Aravinth S. "Design of Efficient Linear Feedback Shift Register for BCH Encoder". In: *Journal of Electrical & Electronic Systems* (Feb. 2015).

[Asa+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. *The landscape of parallel computing research: A view from berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[Ati+13] R. Ben Atitallah, E. Senn, D. Chillet, M. Lanoe, and D. Blouin. "An Efficient Framework for Power-Aware Design of Heterogeneous MPSoC". In: *IEEE Transactions on Industrial Informatics* (Feb. 2013).

[B+96] Grady Booch, Ivar Jacobson, James Rumbaugh, et al. "The unified modeling language". In: *Unix Review* (1996).

[Bai+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga.

The nas parallel benchmarks. Tech. rep. The International Journal of Supercomputer Applications, 1991.

[Bal97] Felice Balarin. *Hardware-software co-design of embedded systems: the POLIS approach*. Springer Science & Business Media, 1997.

[Bas04] Cedric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *13th International Conference on Parallel Architectures and Compilation Techniques*. Sept. 2004.

[BC] Fabrice Bellard and Contributors. *Official QEMU git repository*. `http://git.qemu.org/qemu.git/`.

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Annual Conference on USENIX*. Apr. 2005.

[Ber] U. of Berkeley (USA). *The Spice page*. `http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/`.

[BGS94] David F Bacon, Susan L Graham, and Oliver J Sharp. "Compiler transformations for high-performance computing". In: *ACM Computing Surveys (CSUR)* (Dec. 1994).

[Bie+08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Oct. 2008.

[Bin+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator". In: *SIGARCH Computer Architecture* (May 2011).

[Bla+09] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up, Second Edition*. Springer Publishing Company, Incorporated, 2009.

[Blu+07] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, and T.G. Noll. "Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures". In: *Journal of Systems Architecture* (Jan. 2007).

[Bon+08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. June 2008.

[Bou+08] B. Bougard, B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins. "A Coarse-Grained Array Accelerator for Software-Defined Radio Baseband Processing". In: *IEEE Micro* (July 2008).

[Bra00] C. Brandolese. "A Codesign Approach to Software Power Estimation for Embedded Systems". PhD thesis. Politecnico di Milan, 2000.

[BSN04] H. Blume, M. Schneider, and T.G. Noll. "Power estimation on a functional level for programmable processor". In: *TI Devel Conference*. 2004.

[BSW15] Kirk M. Bresniker, Sharad Singhal, and R. Stanley Williams. "Adapting to Thrive in a New Economy of Memory Abundance". In: (Dec. 2015).

[BTM00] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: a Framework for Architectural-level Power Analysis and Optimizations". In: *International Symposium on Computer Architecture (ISCA)*. June 2000.

[CCN06] Stephen L Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in SCILAB*. 2006.

[Cen+08] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. "MAPS: An Integrated Framework for MPSoC Application Parallelization". In: *45th Annual Design Automation Conference*. June 2008.

[Chu+10] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?" In: *43rd Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2010.

[CM12a] Daniel Cordes and Peter Marwedel. “Multi-objective aware extraction of task-level parallelism using genetic algorithms”. In: *Design, Automation Test in Europe Conference and Exhibition (DATE)*. Mar. 2012.

[CM12b] Daniel Cordes and Peter Marwedel. “PaxES: PArallelism eXtraction for Embedded Systems: Three approaches one tool”. In: *Research Poster at The Designing for Embedded Parallel Computing Platforms (DEPCP)*. Mar. 2012.

[CMA10] Daniel Cordes, Peter Marwedel, and Mallik Arindam. “Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming”. In: *IEEE/ACM/IFIP 8th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Oct. 2010.

[Con+15] J. Cong, Z. Fang, M. Gill, and G. Reinman. “PARADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration”. In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2015.

[Cor+11] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. “Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming”. In: *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2011.

[Cor+12] Daniel Cordes, Michael Engel, Peter Marwedel, and Olaf Neugebauer. “Automatic extraction of multi-objective aware pipeline parallelism using genetic algorithms”. In: *8th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*. Oct. 2012.

[Cor+13a] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. “Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-Core Platforms”. In: *16th International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*. Oct. 2013.

[Cor+13b] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. “Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs”. In: *14th International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*. Oct. 2013.

[Cor14] Daniel Cordes. “Automatic Parallelization for Embedded Multi-Core Systems using High-Level Cost Models”. PhD thesis. dortmund, 2014.

[CR96] T. Chou and K. Roy. “Accurate estimation of power dissipation in cmos sequential circuits”. In: *IEEE Transaction VLSI System*. 1996.

[Cro+14] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[Del+16] Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le Gal, and Christophe Jego. “QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0”. In: *8th European Congress on Embedded Real Time Software and Systems (ERTS)*. Jan. 2016.

[Den+74] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* (Oct. 1974).

[Din+13] B. D. de Dinechin, R. Ayrignac, P. E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. “A clustered many-core processor architecture for embedded and accelerated applications”. In: *IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2013.

[Din13] Benoit Dupont de Dinechin. “Dataflow language compilation for a single chip massively parallel processor”. In: *IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*. Sept. 2013.

[DM12] Julien DeAntoni and Frédéric Mallet. "TimeSquare: Treat Your Models with Logical Time". In: *50th International Conference on Objects, Models, Components, Patterns (TOOLS)*. May 2012.

[DT01] Philips Electronic Design and Philips Research Tools Group. *DIESEL*. 2001.

[Erb07] Cagkan Erbas. *System-level modelling and design space exploration for multiprocessor embedded system-on-chip architectures*. Amsterdam University Press, 2007.

[Esm+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *38th Annual International Symposium on Computer Architecture (ISCA)*. San Jose, California, USA, June 2011, pp. 365–376. ISBN: 978-1-4503-0472-6.

[Flo+13] Antoine Floch, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic L'Hours, Nicolas Simon, Steven Derrien, François Charot, Christophe Wolinski, and Olivier Sentieys. "GeCoS: A framework for prototyping custom hardware design flows". In: *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Sept. 2013.

[FMS08] Sanford Friedenthal, Alan Moore, and Rick Steiner. "OMG Systems Modeling Language (OMG SysML™) Tutorial". In: *INCOSE international symposium*. 2008.

[FOW87] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. "The program dependence graph and its use in optimization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (July 1987).

[Gal62] Robert Gallager. "Low-density parity-check codes". In: *IRE Transactions on information theory* (Jan. 1962).

[Goo+13] Sven Goossens, Benny Akesson, Martijn Koedam, Ashkan Beyranvand Nejad, Andrew Nelson, and Kees Goossens. "The CompSOC Design Flow for Virtual Execution Platforms". In: *10th FPGAworld Conference*. Sept. 2013.

[Gou+11] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. "ΣC: A Programming Model and Language for Embedded Manycores". In: *11th International Conference Algorithms and Architectures for Parallel Processing (ICA3PP)*. Oct. 2011.

[GP94] Milind Girkar and Constantine D. Polychronopoulos. "The hierarchical task graph as a universal intermediate representation". In: *International Journal of Parallel Programming* (1994).

[Gra] Mentors Graphics. *Lsim power analyst : Transistor-level simulation*.

[GS08] Sébastien Gérard and Bran Selic. "The UML – MARTE Standardized Profile". In: *IFAC Proceedings Volumes* (2008).

[Gus+10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. "The Mälardalen WCET benchmarks: Past, present and future". In: *OASIcs-OpenAccess Series in Informatics*. Apr. 2010.

[Han+09] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. "CoMPSoC: A Template for Composable and Predictable Multi-processor System on Chips". In: *ACM Trans. Des. Autom. Electron. Syst.* (Jan. 2009).

[Har+08] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. "Chstone: A benchmark program suite for practical c-based high-level synthesis". In: *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. June 2008.

[Hen99] Jörg Henkel. "A Low Power Hardware/Software Partitioning Approach for Core-based Embedded Systems". In: *36th Annual ACM/IEEE Design Automation Conference (DAC)*. 1999.

[Hua+95] Charlie X. Huang, Bill Zhang, An-Chang Deng, and Burkhard Swirski. "The Design and Implementation of PowerMill". In: *International Symposium on Low Power Design (ISLPED)*. 1995.

[IC] Edgar E. Iglesias and Contributors. *Transaction Level eMulator (TLMu) git repository*. https://github.com/edgarigl/tlmu.

[ILG10] Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. "Parmibench-an open-source benchmark for embedded multiprocessor systems". In: *IEEE Computer Architecture Letters* (Aug. 2010).

[Inc15] Xilinx Inc. *Vivado Design Suite - HLX edition*. 2015. URL: http://www.xilinx.com/products/design-tools/vivado.html.

[Inc16] Xilinx Inc. *Zynq-7000 All Programmable SoC: Technical Reference Manual*. Sept. 2016. URL: http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[Inr] Cairn Inria. *GeCoS : Generic Compiler Suite*. http://gecos.gforge.inira.fr.

[Iri91] François Irigoin *et al. PIPS: Automatic Parallelizer and Code Transformation Framework*. 1991. URL: http://pips4u.org/.

[JMF01] Guohua Jin, J. Mellor-Crummey, and R. Fowler. "Increasing Temporal Locality with Skewing and Recursive Blocking". In: *Supercomputing, ACM/IEEE 2001 Conference*. Nov. 2001.

[Joh09] John Aynsley et al. *OSCI TLM-2.0 Language reference manual*. Open SystemC Initiative (OSCI), July 2009.

[Kah74] Gilles Kahn. "The Semantics of Simple Language for Parallel Programming." In: *International Federation for Information Processing (IFIP)*. Aug. 1974.

[Khe+14] A. Khecharem, C. Gomez, J. Deantoni, F. Mallet, and R. D. Simone. "Execution of heterogeneous models for thermal analysis with a multi-view approach". In: *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*. Oct. 2014.

[KM77] Gilles Kahn and D. B. MacQueen. "Coroutines and networks of parallel processes". In: *International Federation for Information Processing (IFIP)*. Aug. 1977.

[Lau+04] J. Laurent, N. Julien, E. Senn, and E. Martin. "Functional Level Power Analysis: An Efficient Approach for Modeling the Power Consumption of Complex Processors". In: *Design, Automation Test in Europe Conference and Exhibition (DATE)*. Feb. 2004.

[LBN05] J. von Livonius, H. Blume, and T.G. Noll. "FLPA-based power modeling and power aware code optimization for a Trimedia DSP". In: *ProRISC-Workshop*. 2005.

[LC10] Rainer Leupers and Jeronimo Castrillon. "MPSoC Programming Using the MAPS Compiler". In: *Asia and South Pacific Design Automation Conference (ASPDAC)*. Jan. 2010.

[LH98] Yanbing Li and J. Henkel. "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems". In: *35th Annual ACM/IEEE Design Automation Conference (DAC)*. June 1998.

[Li+09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures". In: *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2009.

[Mac99] D. J.C. MacKay. "Good Error-correcting Codes Based on Very Sparse Matrices". In: *IEEE Trans. Inf. Theor.* (Mar. 1999).

[MCB09] M. Montón, J. Carrabina, and M. Burton. "Mixed simulation kernels for high performance virtual platforms". In: *Forum on Specification Design Languages (FDL)*. Sept. 2009.

[Mel] Mellanox technologies. *TILE-Gx72 Processor*. http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf.

[Mit93] Joseph Mitola. "Software radios: Survey, critical evaluation and future directions". In: *IEEE Aerospace and Electronic Systems Magazine* (Apr. 1993).

[Moo65] Gordon E Moore. "Cramming more components onto integrated circuits". In: *IEEE Solid-State Circuits Society Newsletter* (Apr. 1965).

[Nac08] M. Moudgill J. Glossner S. Agrawal G. Nacer. "The sandblaster 2.0 architecture and SB3500 implementation". In: *Software Defined Radio Technical Forum* (Oct. 2008).

[Opt16] Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: `http://www.gurobi.com`.

[Ott+05] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. "Automatic Thread Extraction with Decoupled Software Pipelining". In: *38th Annual IEEE/ACM International Symposium on Microarchitecture*. Nov. 2005.

[Pal+10] Martin Palkovic, Praveen Raghavan, Min Li, Antoine Dejonghe, Liesbet Van der Perre, and Francky Catthoor. "Future Software-Defined Radio Platforms and Mapping Flows". In: *IEEE Signal Processing Magazine* (Mar. 2010).

[PEP06] A. D. Pimentel, C. Erbas, and S. Polstra. "A systematic approach to exploring embedded system architectures at multiple abstraction levels". In: *IEEE Transactions on Computers* (Feb. 2006).

[PHB13] James Pallister, Simon Hollis, and Jeremy Bennett. "BEEBS: Open benchmarks for energy measurements on embedded platforms". In: *arXiv preprint arXiv:1308.5174* (Sept. 2013).

[Pim+01] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. "Exploring Embedded-Systems Architectures with Artemis". In: *Computer* (Nov. 2001).

[Pot+01] Nachiketh R. Potlapally, Anand Raghunathan, Ganesh Lakshminarayana, Michael S. Hsiao, and Srimat T. Chakradhar. "Accurate Power Macro-modeling Techniques for Complex RTL Circuits". In: *Int. Conf. VLSI Design*. Jan. 2001.

[PPL95] Thomas M. Parks, José Luis Pino, and Edvard A. Lee. "A Comparison of Synchronous and Cyclo-Static Dataflow". In: *Asilomar Conference on Signals, Systems and Computers*. Oct. 1995.

[Qu+00] Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak. "Function-level Power Estimation Methodology for Microprocessors". In: *37th Annual Design Automation Conference (DAC)*. June 2000.

[Ram+08] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. "Parallel-stage Decoupled Software Pipelining". In: *6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Apr. 2008.

[Ram07] U. Ramacher. "Software-Defined Radio Prospects for Multistandard Mobile Phones". In: *Computer* (Oct. 2007).

[Rea+14] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. "Machsuite: Benchmarks for accelerator design and customized architectures". In: *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. Oct. 2014.

[Ret+14a] S. K. Rethinagiri, O. Palomar, A. Cristal, O. Unsal, and M. M. Swift. "DESSERT: DESign Space ExploRation Tool based on power and energy at System-Level". In: *27th IEEE International System-on-Chip Conference (SOCC)*. Sept. 2014.

[Ret+14b] S. K. Rethinagiri, O. Palomar, J. Arias Moreno, O. Unsal, and A. Cristal. "VPPET: Virtual platform power and energy estimation tool for heterogeneous MPSoC based FPGA platforms". In: *24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Sept. 2014, pp. 1–8.

[Rou+16] Baptiste Roux, Matthieu Gautier, Olivier Sentieys, and Steven Derrien. "Communication-Based Power Modelling for Heterogeneous Multiprocessor Architectures". In: *IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (mcSoC)*. Sept. 2016.

[Rou+17] Baptiste Roux, Matthieu Gautier, Olivier Sentieys, and Jean-Philippe Delahaye. ""Fast and Energy-Driven Design Space Exploration for Heterogeneous Architectures"". In:

IEEE 25th International Symposium on Field-Programmable Custom Computing Machines (FCCM), Poster. May 2017.

[Sar93] Vivek Sarkar. "A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs". In: *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*. Aug. 1993.

[SC01] A. Sinha and A. P. Chandrakasan. "JouleTrack-a Web based tool for software energy profiling". In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. June 2001.

[Sci12] Scilab Enterprises. *Scilab: Le logiciel open source gratuit de calcul numérique*. Scilab Enterprises. Orsay, France, 2012. URL: `http://www.scilab.org`.

[Sen+05] Eric Senn, Johann Laurent, Nathalie Julien, and Eric Martin. "Softexplorer: Estimating and Optimizing the Power and Energy Consumption of a C Program for DSP Applications". In: *EURASIP J. Appl. Signal Process.* (Jan. 2005).

[Sha+15] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. "The aladdin approach to accelerator design and modeling". In: *IEEE Micro* (May 2015).

[Str+13] Timo Stripf, Oliver Oey, Thomas Bruckschloegla, Juergen Becker, Gerard Rauwerda, Kim Sunesen, George Goulas, Panayiotis Alefragisc, Nikolaos Voros S., Steven Derrien, Olivier Sentieys, Nikolaos Kavvadias, Grigoris Dimitroulakos, Kostas Masselos, Dimitrios Kritharidis, Nikolaos Mitas, and Thomas Perschke. "Compiling Scilab to high performance embedded multicore systems". In: *Microprocessors and Microsystems* (Nov. 2013).

[Syn] Synopsys. *Power-Gate(TM) : a dynamic, simulation-based, gate-level power analysis tool.* `http://www.synopsys.com`.

[Tiw+96] V. Tiwari, S. Malik, A. Wolfe, and M. T. C. Lee. "Instruction level power analysis and optimization of software". In: *9th International Conference on VLSI Design (VLSI)*. Jan. 1996.

[Tor+12] Massimo Torquati, Marco Vanneschi, Mehdi Amini, Serge Guelton, Ronan Keryell, Vincent Lanore, François-Xavier Pasquier, Michel Barreteau, Rémi Barrère, Claudia-Teodora Petrisor, et al. "An innovative compilation tool-chain for embedded multi-core architectures". In: *Embedded World Conference*. 2012.

[Tou11] Georgios Tournavitis. "Profile-driven parallelisation of sequential programs". PhD thesis. Institute of Computing Systems Architecture, School of Informatics, University of Edinburgh, 2011.

[Vac+07] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. "Speculative Decoupled Software Pipelining". In: *16th International Conference on Parallel Architecture and Compilation Techniques*. Sept. 2007.

[Ver10] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software – ICMS 2010: Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*. Springer Berlin Heidelberg, Sept. 2010, pp. 299–302.

[WL91] M. E. Wolf and M. S. Lam. "A Loop Transformation Theory and an Algorithm to Maximize Parallelism". In: *IEEE Trans. Parallel Distrib. Syst.* (Oct. 1991).

[Woh+08] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. "From SODA to scotch: The evolution of a wireless baseband processor". In: *41st IEEE/ACM International Symposium on Microarchitecture*. Nov. 2008.

[WS13] L. Wang and K. Skadron. "Implications of the Power Wall: Dim Cores and Reconfigurable Logic". In: *IEEE Micro* (Sept. 2013).

[WS16] L. Wang and K. Skadron. “Lumos+: Rapid, pre-RTL design space exploration on accelerator-rich heterogeneous architectures with reconfigurable logic”. In: *IEEE 34th International Conference on Computer Design (ICCD)*. Oct. 2016.

[XI] Xilinx and Edgar E. Iglesias. *libsystemctlm-soc git repository*. `https://github.com/Xilinx/libsystemctlm-soc`.

[Xil06] I Xilinx. “Microblaze processor reference guide”. In: *reference manual* (2006).

[Ye+00] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. “The design and use of simplePower: a cycle-accurate energy estimation tool”. In: *37th Design Automation Conference (DAC)*. June 2000, pp. 340–345.

[Zon+11] Ziliang Zong, Adam Manzanares, Xiaojun Ruan, and Xiao Qin. “EAD and PEBD: two energy-aware duplication scheduling algorithms for parallel tasks on homogeneous clusters”. In: *IEEE Transactions on Computers* (Oct. 2011).

List of Figures

List of Tables

ACRONYMES

ADC Analog to Digital Converter
AHTG Augmented Hierarchical Task Graph
ALU Arithmetic-Logic Unit
AMBA Advanced Micro-controller Bus Architecture
APDG Augmented Program dependence Graph
ASIC Application Specific Integrated Circuit
ASIP Application Specific Instruction Processors
BCH Bose-Chaudhuri-Hocquenghem
BRAM Block Random Access Memory
CDFG Contral Data Flow Graph
CFG Control Flow Graph
CGRA Coarse Grain Reconfigurable Architecture
CMOS Complementary Metal Oxide Semi-conductor
CPU Central Processing Unit
DDR Double Data Rate memory
DFG Data Flow Graph
DMI Direct Memory Interface
DSE Design Space Exploration
DSWP Decoupled Software Pipelining
DVB-S Digital Video Broadcasting-Satellite
EDA Electronic Design Automation
EPDG Energy Program Dependence Graph
FLPA Functionnal-level Power Analysis
FPGA Field Programmable Gate Array
FPU Floating Point Unit
GA Genetic Algorithm
HDL Hardware Description Language
HEPC High-Performance Embedded Computing
HLPA Hybrid functional-level and instruction-Level Power Analysis
HLS High Level Synthesis
HMpSoC Heterogeneous Multi-processor System on Chip
HPC High Performance Computing
HTG Hierarchical Task Graph
ILP Integer Linear Programming
ILPA Instruction Level Power Analysis
IPC Inter-Process Communication
IR Intermediate Representation
ISA Instruction Set Architecture
ISS Instruction Set Simulator
JIT Just In Time

KPN Khan Process Network
LDPC Low Density Parity Check
LFSR Linear Feedback Shift Register
LSR Least Square Root
MAC Multiple Access Control
MAU Multiply-Accumulate Unit,
MILP Mixed Interger Linear Programming
MoC Model of Computation
MPI Message Passing Interface
MpSoC Multi-processor System on Chip
NoC Network on Chip
OSCI Open SystemC Initiative
PDG Program Dependence Graph
PL Programmable Logic
QEMU Quick EMUlator
QoS Quality of Service
RTL Register Transfer Level
RUU Register Unit Update
SCC Strongly Connected Component
SDFG Synchronous Data Flow Graph
SDR Software-Defined Radio
SMP Symmetric Multi-Processor
TCG Tiny Code Generator
TLB Transition Look-aside Buffer
UML Unified Modelling language
VLSI Very-Large-Scale Integration
WSCDFG Weighted Statement Control Data Flow Graph